Dynamic Scheduling with Work Stealing - Framework and Analysis of Different Strategies

Adarsh Shreedhar (adarshsr), Bhakti Chaudhari (bchaudha)

Summary

We implemented a Task-Scheduling Framework with Work Stealing, and explored various parameters wrt granularity, nature of the benchmark, Stealing Logic, Queue Logic and gathered analysis of the results obtained. For unevenly-distributed benchmarks, our work-stealing framework provides a clear speedup, and for evenly-distributed benchmarks, we observe that work stealing performs slower than statically distributed workloads.

We used PSC machines to gather our experimental data. Our deliverables were -

- Implement a task scheduling library that provides APIs for various scheduling algorithms, each of which employ different data-structures to store the tasks, as well as different strategies in stealing work during cases of uneven load distribution.
- We will employ three data structures – a centralized queue, a per-task queue and priority queue. We will then run these algorithms for various datasets and problem definitions, on different execution unit combinations, and collect and analyze the speedup obtained.

Background

In Multithreaded or Multiprocessor Computations, Work Stealing is a way of balancing the workload across different execution units. After the initial assignment of tasks to threads, due to scheduling, nature of task split or computation, it may happen that certain threads complete faster than others. In such cases, they can steal tasks originally intended for other threads – thereby not only keeping itself busy, but also reducing the workload on the other thread. There are many ways in which we can maintain the list of tasks – a centralized doubly-ended queue, a per-processor or per-thread queue with preassigned tasks, or a priority queue based on certain heuristics. As with a queue, we can choose to remove tasks from the top or bottom. A right set of choices of the implementation details for a workload can greatly improve parallelism, and a wrong set of choices may hinder performance than without any work stealing itself. Moreover, for problems that need a first/single-result logic, the importance of the scheduler becomes even more underlined. We have implemented different workload schedulers with various work stealing strategies.

As part of our literature review, we noted the following important points that we have tried to incorporate -

- [2] presents an interesting idea of confidence-based work stealing, wherein they deal with a problem of search i.e. an O(1) result by parallelly searching an O(n) space. We would like to implement our priority queue-based implementation based on this idea, i.e., to use a set of heuristics to change the priority and schedule work so as to find the result as quickly as possible.
- [3] attempts to explore randomized work stealing on large scale systems through use of Partitioned Global Address Space (PGAS). While the implementation is out of scope for our survey, they explore various benchmarks and make use of split task queues and lockless release operations, which we believe can prove to be an important resource for perusal.
- [4] is a survey paper that reviews work stealing scheduling from the perspective of scheduling algorithms, optimization of algorithm implementation and processor architecture-oriented optimization.
- [6] introduces Wool, a work-stealing library that is very similar to Cilk, but allows the user to make these calls in C itself, without having to write in a different language or construct (Cilk-5 or Intel TBB). Since we wanted to provide a simple interface abstracting the multiple implementations of work stealing algorithms, this provided valuable information.

Design

The core components of our design are -

1. Thread Pool

Creating a pool of threads before we schedule the tasks helps avoid expensive runtime costs in creating and deleting address-space/ signal-set and other resources for each task. This also helps reuse the threads.

In our design we initialize the threadpool based on the number of threads passed by user. We start the threads with a worker function whose job is to busy wait on the scheduling queue until it gets a task and then run it. Once all the tasks are done, the threadpool will clean up all its threads and terminate.

2. The Scheduler

The initial distribution of the tasks ensures that all processors/execution units are busy. A basic rule of thumb is to create at least as many tasks as there are a number of threads, which should be at least as many as the execution units available.

We have three different schedulers in our implementation -

a. threadpool_centralised.cpp: For centralized task queue
b. threadpool_perthread.cpp: For individual task queues per thread.
c. threadpool_prority.cpp: For priority based scheduling

3. The Work Stealing Algorithm

Since there is a communication penalty in stealing a task, the algorithm must factor whether the currently executing thread would still be occupied by the time the tasks have been brought to the new unit's space completely and begin execution. Depending on the task granularity and the number of tasks available for stealing, we must also consider the number of tasks to steal – since it is faster to steal multiple tasks at once than at different intervals. It also depends on whether we steal multiple tasks from the same thread or individual tasks from various threads.

In our design we support the following types of work stealing -

a. Steal one task from next non empty neighboring thread (thread with threadid+1)
b. Steal half the tasks from next non empty neighboring thread
c. Steal all tasks from next non empty neighboring thread
d. Steal one task from a random non empty thread
e. Steal half the tasks from a random non empty thread
f. Steal all tasks of a random non empty thread

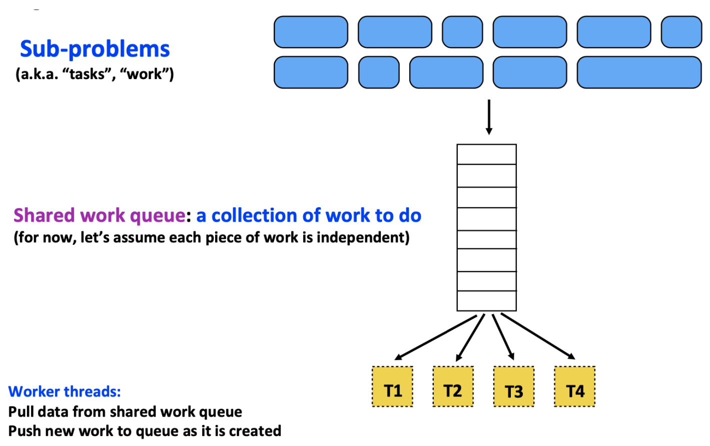
4. The Benchmarking Framework

To benchmark the above scenarios on different use cases, we created a simple benchmarking framework which is responsible for setting the inputs, dividing the work into a given number of tasks and defining the worker method of the specific use case.

The base interface used for developing these benchmarks is benchmark.h.

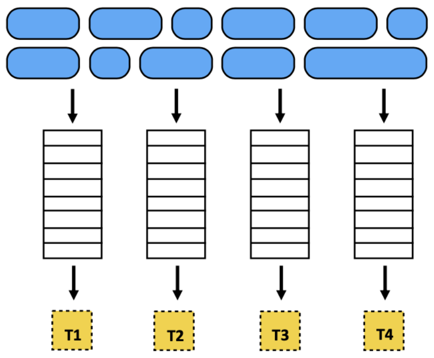
Approach

We first implemented a single queue and a pool of threads, each of which would run a busy-wait loop on the queue on its creation. The user would then split the workload into smaller granularities based on the number of tasks, and then submit them to the queue. After adding all the tasks, the user can then dispatch the queue, so that they start dequeuing the tasks and executing them. Since there is a single resource (queue), contention between threads can arise and therefore we add locks for enqueue and dequeue operations.

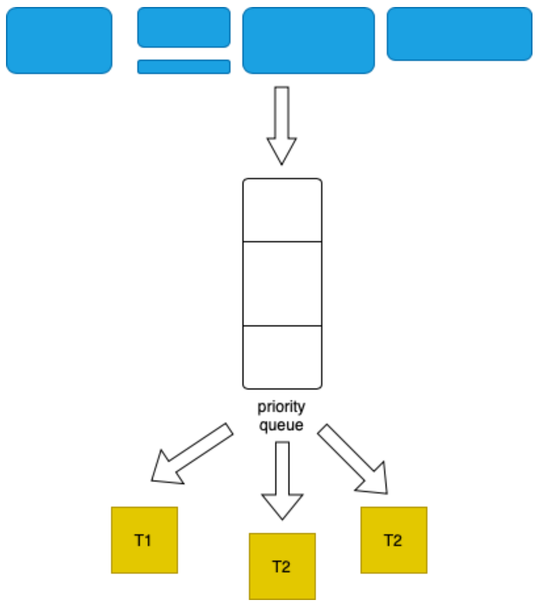

Sub-problems
(a.k.a. "tasks", "work")
Shared work queue: a collection of work to do
(for now, let's assume each piece of work is independent)
Worker threads:
Pull data from shared work queue
Push new work to queue as it is created
T1
T2
T3
T4

Source: lecture slides

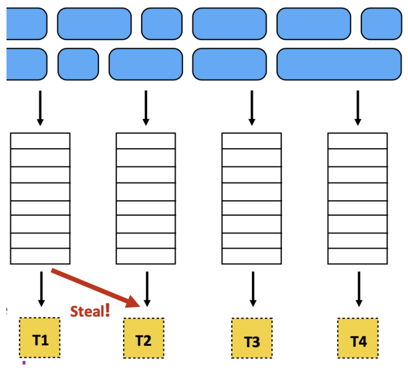
We then moved onto creating queues for each thread, and distributing the tasks across the queues, so that each thread can only pick tasks from its own queue and remove contending for the centralized queue. We added flexibility of the user submitting all the tasks at once, and internally handling distributing the tasks. We used round-robin logic to ensure all queues get similar task numbers, without any knowledge on how long each task may take (i.e. unaware of the workloads being distributed equally).



We then added a priority queue type, that can stop executing all threads once the result is found. This can arrange priority that is set by the user, so that if the result is likely available in a subset of tasks, then prioritize running them first. This was added to aid work that has a single-result type, as against all of the workload to be processed to obtain the result.



We then added the work-stealing logic to the per-thread queue, wherein, the threads then begin searching for tasks in other queues and steal them. The logic is such that if the queue is already empty, we move to the next queue, until we find tasks to steal from. Since this would bring contention again, we then added locks to queue operations - resulting in two queue implementations, with and without locking.



Since we store our thread queues in a vector, it was easy for each thread to check the other thread's queue and steal tasks. Our first stealing logic included iterating over the rest of the queue to find queues that are non-empty and then pop execute them, however we also added randomly choosing the queue to steal tasks. These are now both available as parameters for the user while running the benchmark.

We then added a granularity parameter of the number of tasks to steal from the queue. We can either steal one task, half the tasks available in the victim's queue or all of the tasks in that queue. These are also exposed as parameters, and depending on the nature of the workload can give varied performance, as can be seen in the benchmarks.

To Summarize, our framework provides the flexibility:

1. Running Tasks on a Centralized, Per-Thread or Priority Queue
2. Running Tasks with and without Work Stealing
3. Changing the Nature of the Work Stealing - between Random Selection and Nearest Neighbor.
4. Changing the amount of tasks to steal from the victim queue - between one task, all tasks, and half the tasks.

Benchmarks

We have designed 4 benchmarks that we will define here. To implement these benchmarks we have created an interface benchmark.h which is defined below. Each benchmark defined extends this class and implements the below methods.

```
public:
  // method to set certain input values for this benchmark
  void setInputs() {}

  // method to divide the workload into given number of tasks
  template <typename T>
  void getTasks(T **args, int numberOfTasks) {}

  // worker method which takes a set of arguments in the form of a struct pointer
  void workerTask(void *threadArgs) {}
};
```

1. Mandelbrot

The mandelbrot image generated is a visualization of a famous set of complex numbers called the Mandelbrot set. Each pixel in the image corresponds to a value in the complex plane, and the brightness of each pixel is proportional to the computational cost of determining whether the value is contained in the Mandelbrot set—white pixels required the maximum (256) number of iterations, dark ones only a few iterations, and colored pixels were somewhere in between. This is a good example of unevenly distributed workload as every pixel could require a varying number of iterations.

Set of inputs -

- viewIndex - Different view index values to vary magnification. Greater viewIndex values have more unevenly distributed workloads.

2. Matrix Multiplication

We have implemented parallel matrix multiplication. We have used a very simple algorithm which varies the innermost loop indices based on the task it's working on. This is a good benchmark to represent an uneven workload. Although it should be noted that the number of iterations in each task may be the same, keeping the Big-O complexity constant across tasks, if the elements in the matrix are largely skewed or its a sparse matrix then even this could introduce an uneven workload.

The worker method is defined below

```
// for n*n matrices
for(int i=0; i<n;i++){
        for(int j=0; j<n; j++){
           // vary index based on task
            for(int k=start_idx; k<=end_idx;i++){
                output[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
```

Set of Inputs -

- Matrix dimension: We multiply two n*n matrices

3. Synthetic Benchmark

We chose to implement a synthetic benchmark which simulates an uneven workload. We did this as it allowed us to parameterise the nature of the workload distribution to simulate different kinds of distributions. We do this by putting tasks containing some sleep calls with different values based on our parameters. We send a particular task a sleep timer equivalent to the task ID. So task 1 will sleep for 1s, 2 will sleep for 2s and so on. Since the threads would then execute tasks of vastly different sleep durations, chances are that many threads exhaust their tasks, giving good advantage when employed with work stealing.

4. Text Search (Find First)

We chose a benchmark which represents the "find first" kind of problem. This means that if any one task succeeds then the result of the whole application is found and all other tasks can be terminated. The search problem is the best example of this class of problems. As soon as we find even one occurrence of the target phrase in our search input, we can return true and exit. The performance of such kinds of applications are dramatically affected by the work scheduling logic used. If the task containing the word is run at the end then the application unnecessarily needs to run all the tasks as well. As opposed to if it is scheduled during the beginning, the program can exit early and avoid running unnecessary tasks. The scheduling of tasks can be based on some priority value and we can use a priority queue as our scheduling data structure so that tasks are picked by threads in order of priority.

For the purposes of our benchmarking, we assume that the user has some sort of knowledge base on the search text, which can be a heuristic applicable to the set. This heuristic can give a rough priority ordering of which section of the text the target phrase is likely to be found. For example, consider that you are searching for a "work stealing" in the PCA course material. You may have some idea that this topic was covered in the first half of the semester and hence you are more likely to find this term in the beginning or middle of the text. You can then provide priorities accordingly. This can potentially improve your search performance if the priorities correctly represent the probability of finding the target phrase in a particular section of the text.

We have implemented a very simple text search algorithm which initially tokenises the entire input string and then distributes different sections of the tokenized string to different tasks by index. We are aware that this is not a good real world example as often data will be too large to store in-memory but since our focus is on representing the performance of different types of work scheduling algorithms, we felt this would suffice.

Set of Inputs -

- Search text: Text to be searched
- Target word: Word to search for

Results & Analysis

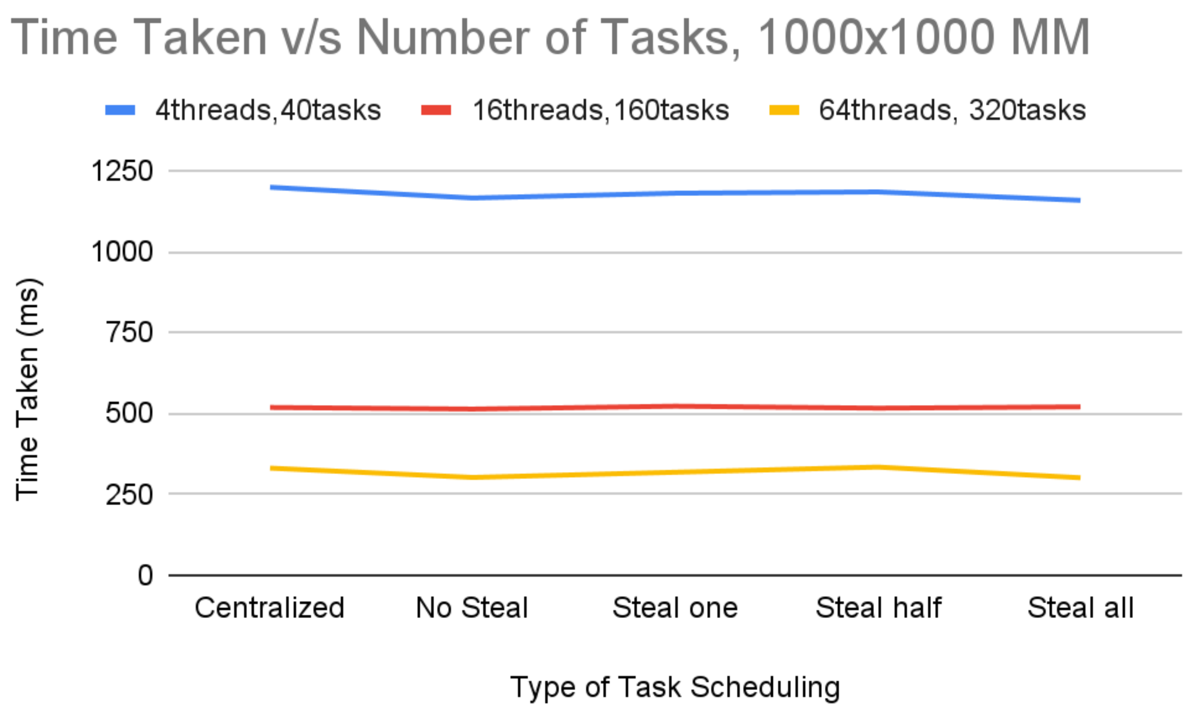
To test our different implementations, we parameterized the following aspects -

- Scheduler type - centralized queue, per thread queue, priority queue
- Stealing strategy - Steal from next neighboring thread, steal from random thread
- Number of tasks to steal - Steal one, half or all tasks from the other queue.
- Input size - For different benchmarks, for ex matrix dimensions for matrix multiplication

We have summarized the most relevant trends that we saw from our analysis below. You can find the complete collected experimental results here. We have collected over 30+ results, given our high flexibility and number of parameters that can be changed.

1. Comparing centralized task queue with parallel work stealing queues in Matrix Mul

The below graph shows the time taken for running matrix multiplication on a 1000X1000 matrix with the given number of threads, tasks and distribution strategy.



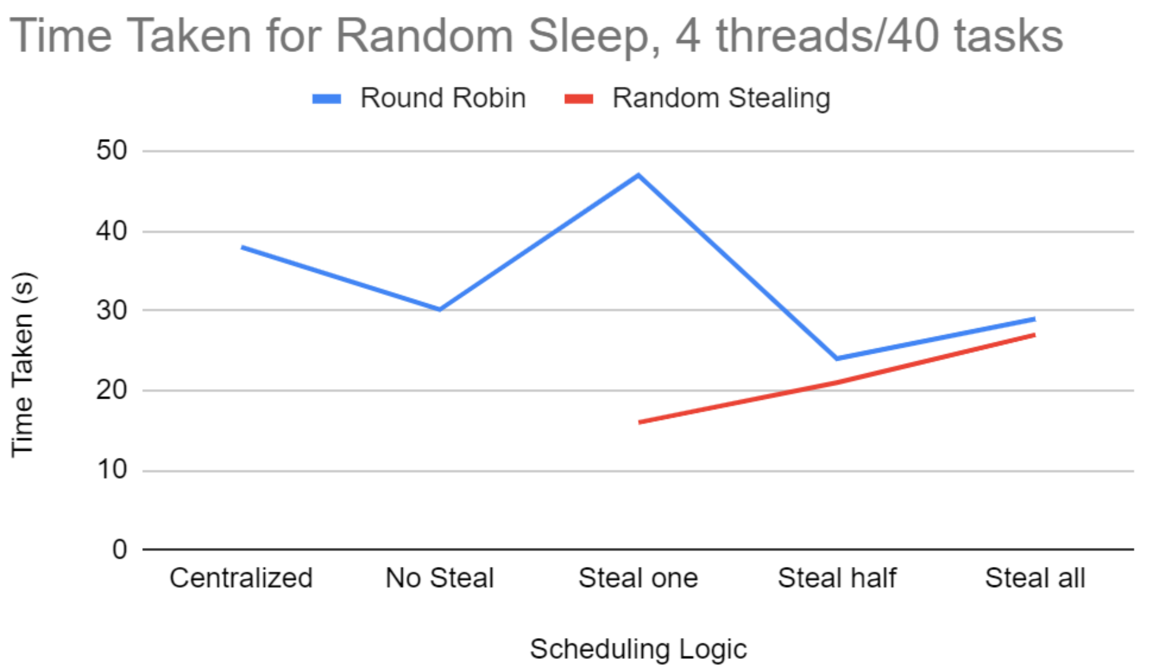
Takeaways from the graph -

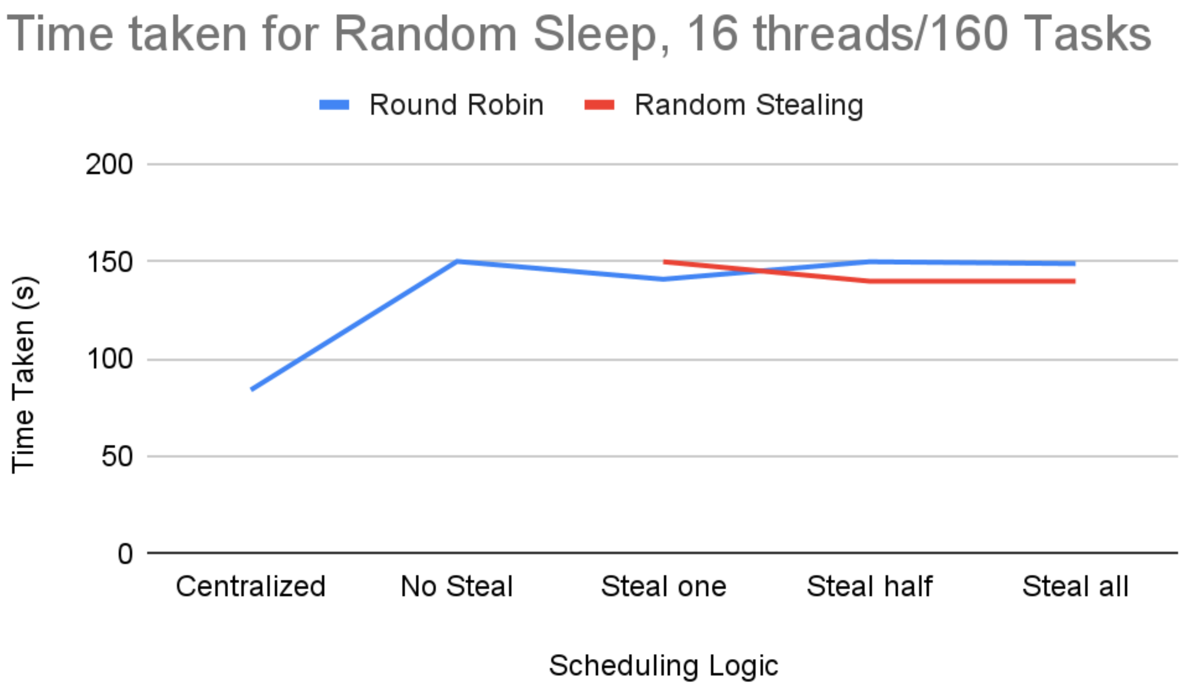
- We can see a slight dip in the time taken for execution in per thread queues (with no stealing) compared to centralized queue. Although the dip is very slight (~10ms), it will become exponential as we increase the input size and task count.
- We don't see a very prominent dip as matrix multiplication is not an inherently uneven workload as each task performs the same number of iterations. Hence, the improvement here comes from skew in number of instructions performed due to the matrix elements being skewed. For ex - some section of the matrix may have very big numbers compared to others, or it may be a very sparse matrix.
- We then see a slight increase/almost same time taken when stealing is implemented. This is because the unevenness of the workload is not significant enough to justify the extra cost of stealing. Stealing requires locking the work queues of other threads and communicating data across processors which can be quite expensive.

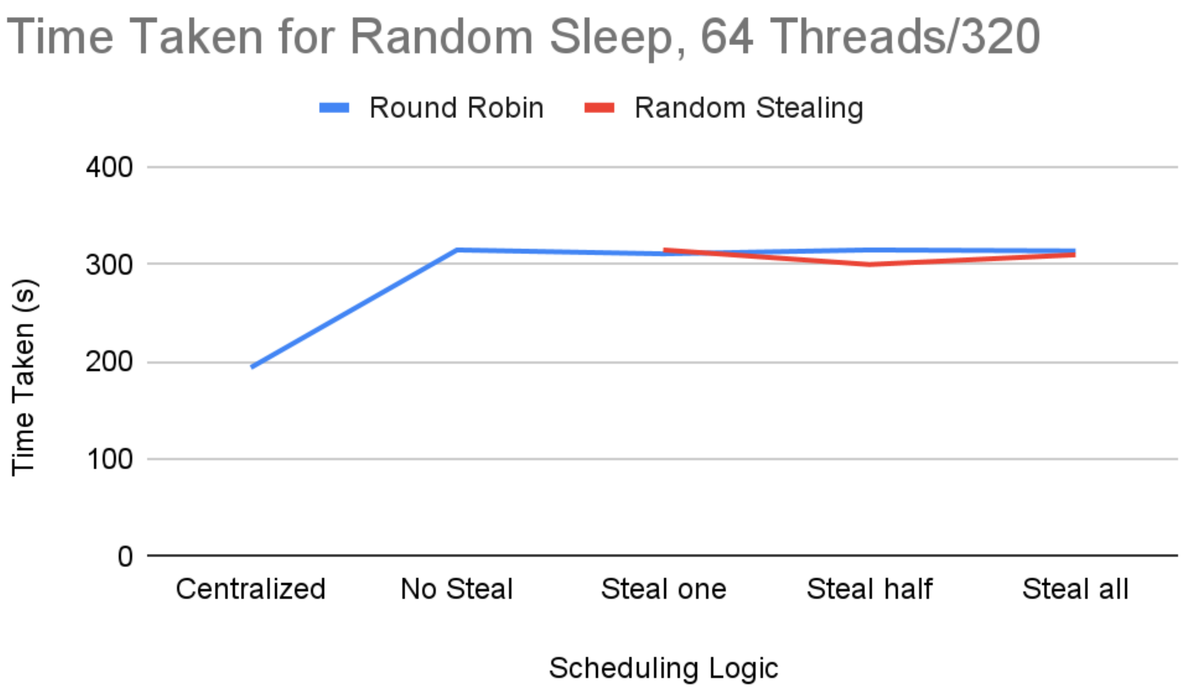
2. Synthetic Benchmark

Due to the nature of the above workload, we could not see the significance that different work stealing algorithms have on the performance. Here we have analyzed the results of our synthetic benchmark.

As our overall workload changes with the change in number of tasks, we have not compared different task counts with each other but instead against different scheduling techniques. Below we have graphs from 40, 160 and 320 tasks.


Time Taken for Random Sleep, 4 threads/40 tasks
Round Robin
Random Stealing
Time Taken (s)
50
40
30
20
10
0
Centralized
No Steal
Steal one
Steal half
Steal all
Scheduling Logic


Time taken for Random Sleep, 16 threads/160 Tasks
Round Robin
Random Stealing
Time Taken (s)
200
150
100
50
0
Centralized
No Steal
Steal one
Steal half
Steal all
Scheduling Logic



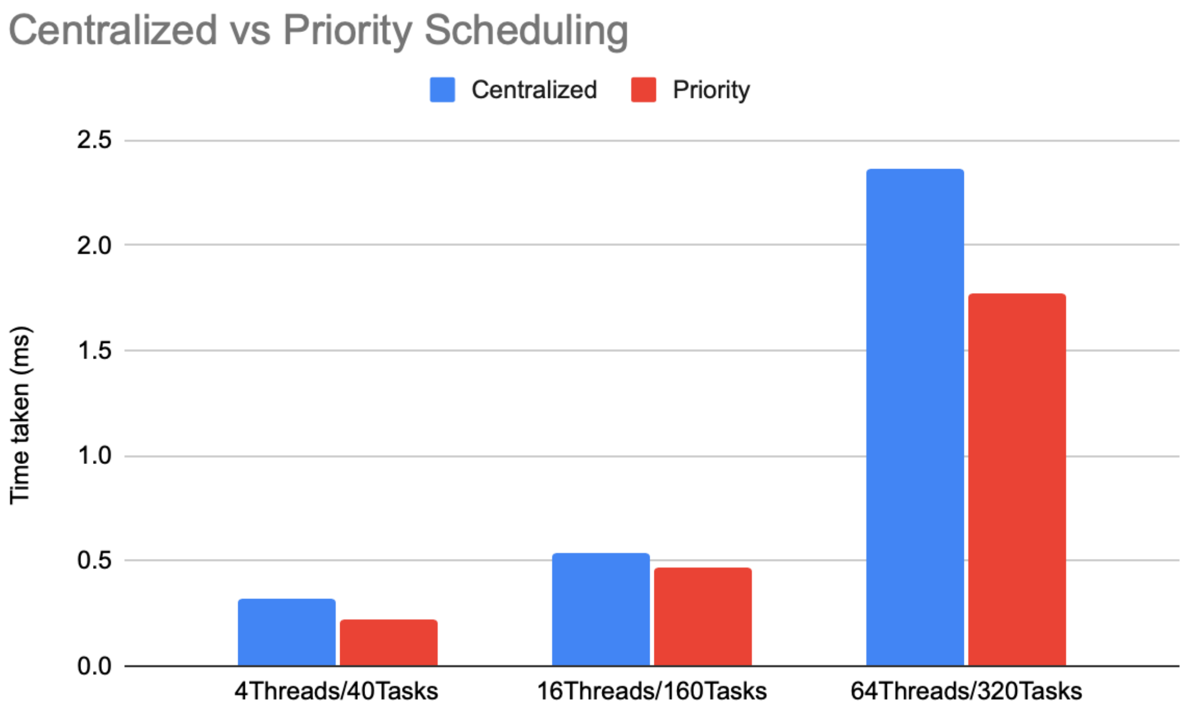
Takeaways from the graphs -

- We see a significant dip when we move from centralized to per thread queues (no stealing). This is because of the low contention for the task queue.
- Once we start stealing (steal one), we see an increase again in time taken, this is because the performance gained by stealing is not justifying the stealing overhead. As we are only stealing one task at a time, this adds a lot of overhead to lock the queue of the victim thread and communicate across processors for each task.
- When we increase the number of tasks we are stealing, we see a significant reduction in time taken (the best performance for round robin). The stealing overhead is now reduced as we lock and communicate once for multiple tasks.
- When we try to steal all tasks, as expected the time taken increases again. This is because we may be getting into a situation of thrashing where each thread is trying to steal the other one's tasks as we empty out task queues when we steal. Somewhere in between stealing half and stealing all we cross the point of diminishing returns.
- Overall we see random stealing performs better than round robin (neighbor) stealing. This is because there may be some bias in the way the tasks are queued. For example, in parallel quicksort the parent task will be the most expensive as it is responsible for sorting the entire array. As it forks child tasks recursively, the size of the array keeps getting smaller. So there is a linearly decreasing intensity of tasks. In such cases stealing from random threads can allow to balance it out.

- As the number of threads/tasks increases across the graphs, we see the above trends less explicitly. This is because we notice an increase in the number of tasks being stolen which causes increased overhead. For ex, in 40 tasks with steal one we saw 3 steals happening vs in 320 tasks we saw 63 tasks being stolen. At this point, we should analyze the workload scheduling itself and optimize for that instead of stealing algorithms.

3. Text Search using priority based scheduling

   We generated a large file with random text and chose a target word for our search. For priority implementation we assigned priorities of the tasks based on the probability of the target word to be present in a certain section of the text.



Takeaways from the graph -
- We can clearly see a reduction in execution time when using priority based scheduling over round robin scheduling used in centralized task queues.
- The difference is not very significant for lower task counts as compared to higher task counts as the overhead of maintaining a priority queue is not justified for lower counts.

Summary & Future Work

We have been successful in implementing and analyzing the impact of various scheduling and work stealing algorithms. We can see that based on the problem statement, the choice of scheduling can have a huge impact on the performance. We think the aspects of this work could be used to build on some of these future ideas -

- Developing a heuristic to automatically choose the best type of scheduling algorithm given a specific input problem.
- Exploring more ways of optimizing the scheduling by also accounting for inter-process communication cost and leverage caching.
- Ways to reduce the overhead of task queue management by trying different more advanced data structures like lock free queues.
- Developing advanced scheduling algorithms that are built specifically for different CPU architectures keeping locality, data movement, cache design and interprocess communication in mind.

References

[1] Blumofe, R.D. and Leiserson, C.E., 1999. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), 46(5), pp.720-748.

[2] Chu G., Schulte C., Stuckey P.J. (2009) Confidence-Based Work Stealing in Parallel Constraint Programming. In: Gent I.P. (eds) Principles and Practice of Constraint Programming - CP 2009. CP 2009. Lecture Notes in Computer Science, vol 5732. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-04244-7_20

[3] SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis November 2009 Article No.: 53Pages 1–11. https://doi.org/10.1145/1654059.1654113

[4] Yang, J., He, Q. Scheduling Parallel Computations by Work Stealing: A Survey. Int J Parallel Prog 46, 173–197 (2018). https://doi.org/10.1007/s10766-016-0484-8

[5] PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming August 1995 Pages 207–216. https://doi.org/10.1145/209936.209958

[6] Faxen, Karl-Filip. (2008). Wool-A work stealing library. SIGARCH Computer Architecture News. 36. 93-100. 10.1145/1556444.1556457.