Optimizing work-stealing

RAJ SAHU, Virginia Tech, USA

ThreadPools provide an efficient way to benefit from multi-threading and multi-processing by removing the overhead of stack allocation for each new thread created. But as a parallel execution progresses, the distribution of work load becomes skewed such that few processors have more tasks compared to others. Task load can be measured by the number of Runnables in ready state within a thread's local deque. Work stealing is an interesting technique where less busier threads steals tasks from more busy ones with an intent to balance the work load throughout the system. In this work, I study some of the algorithms from the literature dealing with work-stealing. After this, I describe about an implementation and later compare its performance with java's inbuilt service along with the other algorithms from the literature.

Additional Key Words and Phrases: **Work Stealing**, **Thread Pool**, **Concurrent Deque**

1 INTRODUCTION

With the development of machines now being focussed more towards leveraging several processors than to run on a single powerful processor, its important to break down a big task into several smaller tasks, efficiently, and run them somewhat independently on different processors. There are two major concerns here: 1) The task breakdown should incur minimal overhead on the system while each task having least dependency on all other remaining tasks; 2) To ensure that these small tasks are uniformly distributed to all available workers (processors) otherwise the system will have a bottleneck due to the overloaded worker. In this work, I will be focussing on the later problem.

Work-Stealing has become a goto approach to load-balance a system where free nodes take away tasks from busy nodes based on some heuristics such as random selection. There has been quiet lot of optimization suggested in the literature. They focus on these broad aspects of work-stealing : Task Granularity, Local Sensitivity and Improvement of Task Queue.

Task Granularity signifies the amount of work each task brings to a worker. If the amount of work is very small i.e fine-grained, the overhead of creating them will overpower the benefit from splitting a task into smaller tasks inorder to run on different processors. On the other hand, coarse-grained tasks can induce imbalance in work distribution. Cao et al. [2011], Hoffmann and Rauber [2008], Tzannes et al. [2010], and Wang et al. [2010] proposed an adaptive approach to ensure the right task granularity. Their work talked mainly about taking into account how much load is there at a given instant in each of taskQueues and then determining whether it would be better to split a new task to evenly distribute it or just run it sequentially on the current thread itself.

Local Sensitivity is about taking into account a system's architecture to decide which thread's tasks should be stolen. To elaborate, if a parent task in running on a given processor then it will be much more efficient to run its child processes in the same processor. The reason behind this is cache-invalidation. In NUMA architechtures, where each core has its own share of memory, if a task is migrated from one core to another, all the required cache will first be invalidated from the previous core, and then reloaded to the new core. Also other factors like false sharing further amplifies the overhead. Acar et al. [2000] and Robison et al. [2008] proposed a mailbox based technique where each thread will have another list (mailbox) along with the taskQueue which contains all those tasks which have affinity with that particular thread. This way, once a thread is free, it can directly start working on one of the tasks from the mailbox rather than stealing tasks from other threads. Chen et al. [2011] and Olivier et al. [2011] proposed having a common

Author's address: Raj Sahu, raj.sahu@vt.edu, Virginia Tech, Blacksburg, Virginia, USA, 24060.

taskQueue for a core such that whenever a thread within this core needs to steal, it will only get tasks from threads within the same core.

Improvement of Task queues is another class of work where the focus is to reduce the data race to steal a task by many threads. Initially, all the threads had a common Queue to get/steal tasks from. This was then improved such that each thread will have a separate Queue which supports Atomic push and pop operations and hence is lock free. Chase and Lev [2005] introduced the unbounded lock-free queue using cyclic queue. Other improvements include : separating a taskQueue into shared and private section [Dinan et al. 2009] and using the hardware to implement the taskQueue [Michael et al. 2009].

In this work, I am interested in the overheads due to frequent attempts of stealing. As pointed out by Qian et al. [2015] there are hundreds of stealing attempts every second due to which if the selection process is not efficient or correct, then a significant amount of CPU cycles is wasted. Not to forget that a stealing attempt is not always successful (about 20% as per the author). In the next section, I will be detailing about the proposed optimization, compare it with existing work-stealing techniques and discuss about the outcomes.

2 DESIGN

The implementation has two legs : one focuses on reducing frequent decision making stealing attempts, while the other one dynamically adjusts the granularity of tasks being pushed into the taskQueues.

2.1 Thread Selection

Whenever a thread finds its taskQueue empty, it goes for stealing. There are several approach one can take to implement stealing : random selection, best of two, best of all. Random selection just iterates through all threads and picks the first available task it can. Best of Two randomly picks two threads and steals a task from the deepest taskQueue out of the two. Best of all is greedy stealing where the global busiest thread is identified and its task is stolen.

In all the above techniques, there is a significant amount of work being done to just determine which thread to steal from. As the stealing is done atleast hundreds of time if not thousands, this selection procedure contributes a lot of baggage in the stealing process. As an optimization, it's better we identify the global busiest thread for once and use this result in subsequent steal() calls until its taskQueue is exhausted. Once the size is reduced to zero, the steal() method can again look for a different victim and start back with this cycle. In this report, I will refer this as *memoStealing*.

2.2 Granularity Adjustment

Granularity can be understood as the amount of work each thread is willing to do; if a worker gets tasks bigger than this threshold, it will simply break it down and push one-half to the pool while process the other-half.

Even with a stealing heuristic in-place, if the rate of task creation is too high, the work-load distribution can get skewed nevertheless. Hence, we want to determine at random instants during execution whether the work-load has become non-uniform and based on that decrease or increase the granularity of tasks. By doing experiments, I observed that reducing the granularity upon skewedness helped improving the performance.

3 IMPLEMENTATION

The complete code is implemented using Java programming language. I first implemented a LockFree unbounded double-ended queue *concurrentDeque* inspired by Chase and Lev [2005]. This data

Table 1. List of parameters used for optimizing Thread Selection

Parameter	Description	Value
steal_algo	Name of the algorithm to use for stealing	firstAvailable, bestOfTwoRandom, bestofAll, memoStealing
numThreads	Number of threads to initiate in threadPool	1-8
Threshold	Granularity level for each task	100

structure internally uses a circular array which gets resized whenever required. *ConcurrentDeque* uses atomic operation *CompareAndSet* to provide concurrency without needing locks.

Next, I implemented a custom ThreadPool with work-stealing algorithms. This class starts *numThreads* number of threads which will keep looking for threads until interrupted. This class has a *submit()* method used by anyone to push a new Runnable task to one of the taskQueues. The *awaitTermination()* method checks whether any of the threads are still having tasks to complete and blocks until all threads are free while the *shutdown()* method first awaits for termination and then sends an interrupt to all the threads which makes them stop taking new tasks.

For a sample task, I chose the problem to be finding the number of prime numbers within a given range. The *PrimeNumbers* class implements Runnable to provide the *compute()* function which is used by our ThreadPool to start execution of a task. Other than functions to perform the basic prime number counter, the class also has a *makeFine()* and *makeCoarse()* function to decrease or increase the global value of *THRESHOLD* by 10%, respectively. These two functions wil be called by the ThreadPool whenever it detects that the thread work-load division is getting very uneven.

In the Main class, we first create an object of our ThreadPool class. Then, in a loop, we create an object of the *PrimeNumbers* and start its execution by explicitly calling the *compute()* function. Once the *awaitTermination()* returns we get the time taken for the iteration. At the end, we get the total runtime and throughput for given set of parameters.

4 EVALUATION

All the calculations has been done on my x86_64 i5-7200U CPU @ 2.50GHz machine with 4 CPUs, 2 threads per code and NUMA enabled.

For the first experiment, we will test the modification done to improve Thread Selection. Table 1 enlists the several parameters used in this experiment. We study various outputs such as runtime, throughput, total time spent per thread on stealing as a percentage of runtime and total steal success for each of the *steal_algo* as we vary the *numThreads*. Figure 1 has four different plots for each of the above mentioned outputs. A glance over the Runtime vs ThreadCount plot informs us that the custom implementation done is having many overheads which are being handled well within the java's inbuilt Executor Service. In terms of runtime and throughput, all the four stealing algorithms perform about the same. In terms of the time spend to steal a task, the proposed algorithm performs reasonably well until we move towards higher thread count. A probable reason for this could be that multiple threads start taking on a single victim which leads to contention between them. From the Steal success graph, the proposed algorithm gives good performance until thread count of 4 (similar to the Time Spent Stealing graph) and then its performance decays quickly. This can too be justified with the previous reasoning. Overall, we can now come to a conclusion that the proposed stealing algorithm works well in medium level of concurrency.

We now move towards our second and final experiment concerning the performance of granularity adjustment optimization. Other than parameters listed in Table 1, Table 2 lists down the other

Fig. 1. Performance charts for the Thread Selection optimization

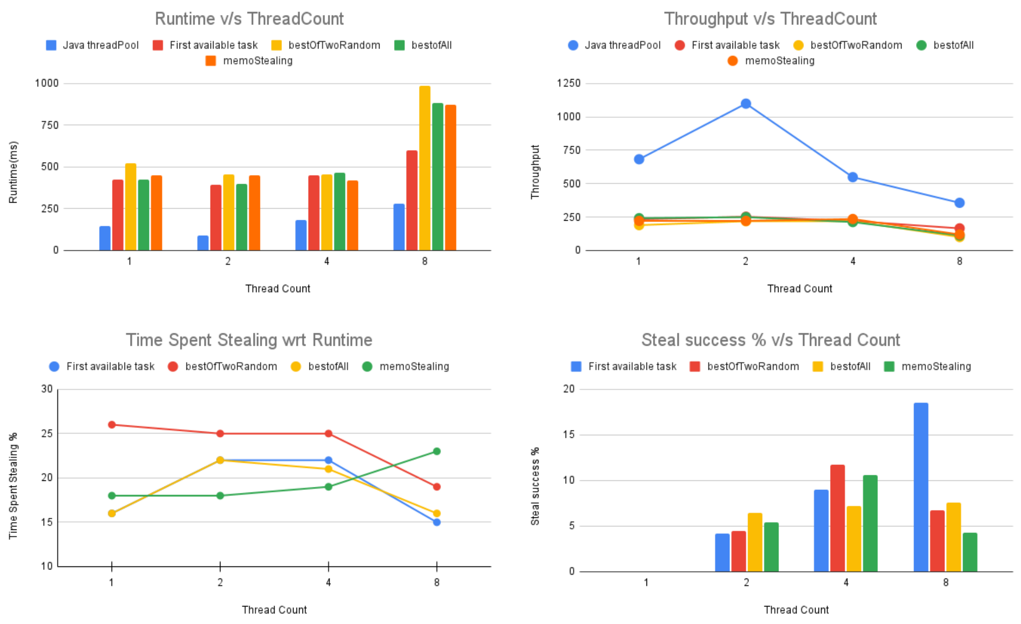


Table 2. List of parameters used for optimizing Granularity Adjustment

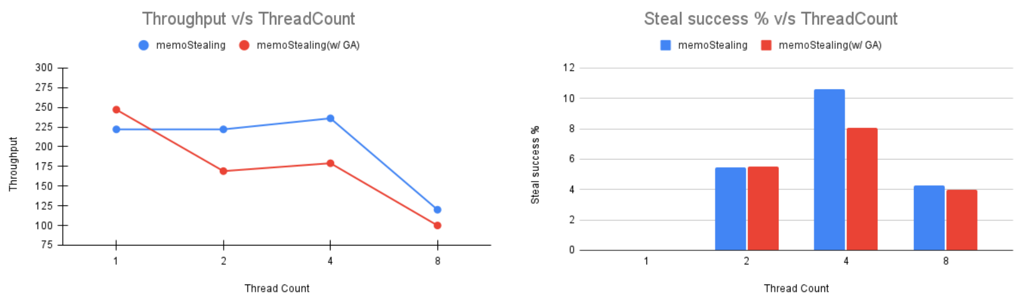
Parameter	Description	Value
probability_checkStdDev	Control the frequency of making adjustments	0.01
stdDev_threshold	Threshold value of deviation to decide when to reduce the Runnable's Threshold	0.5

parameters used for this experiment. We will use the proposed *memoStealing* algorithm and pair it up with the dynamic granularity adjustment described in section 2.2 to see the impact. Figure 2 has two graph plots showing the performance of memoStealing with and without Granularity Adjustments (abbreviated as GA). Inferring from both the plots, we can see that the GA modification is effective for low contention and its performance benefit becomes an overhead quickly as we go towards higher number of threads. A probable reason is that the extra lines of code for calculating standard deviation using depth of all the existing taskQueues before even deciding whether to change the granularity brings alot of baggage which overpowers the benefit due to change in granularity, specially when the number of threads increases. On top of this, the taskQueues are constantly changing, so probably by the time we calculate the standard deviation to make a decision, the taskQueues have already changed. Due to this, the decisions being made became stale and incorrect.

5 CONCLUSION

In this work, I studied about various existing work-stealing algorithms in the literature. Based on the three classes of optimizations : Task Granularity, Local Sensitivity and Task Queue Improvement, I

Fig. 2. Performance charts for the Granularity Adjustment



proposed an optimization for Granularity Adjustment combined with a reduced cost of selection performed at each stealing procedure. The suggested improvement in Thread Selection performed reasonably for medium level of parallelism (upto 4 threads) but couldn't scale to higher thread count probably because all threads, as per the algorithm, starts to target the globally busiest thread which leads to contention. Using this Thread selection procedure, we then evaluated the optimization suggested for Granularity adjustment where the idea was to check the skewedness of task distribution with the help of standard deviation at random instants during the execution and reduce the granularity of the Runnable tasks. As per the observation, this technique performed well in low parallelism environment but the performance benefit quickly became performance baggage due to the code overhead induced for performing each steal() process. At the end, we saw a net gain in performance for low thread count when we use both the suggested modifications.

For future work, we can first aim to bridge the gap between our custom implementation and the Java inbuilt ExecutorService implementation. Once this is achieved, we will be able to truely identify the net performance gain by the proposed modifications. Also, it would be interesting to see combine the proposed method with cache-locality based improvements.

REFERENCES

Umut A Acar, Guy E Blelloch, and Robert D Blumofe. 2000. "The data locality of work stealing." In: *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, 1–12.

Qian Cao, Changjun Hu, Shigang Li, and Haohu He. 2011. "Adaptive task granularity strategy for OpenMP3. 0 task model on cell architecture." In: *International Conference on High Performance Networking, Computing and Communication Systems*. Springer, 393–400.

David Chase and Yossi Lev. 2005. "Dynamic circular work-stealing deque." In: *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, 21–28.

Quan Chen, Zhiyi Huang, Minyi Guo, and Jingyu Zhou. 2011. "Cab: Cache aware bi-tier task-stealing in multi-socket multi-core architecture." In: *2011 International Conference on Parallel Processing*. IEEE, 722–732.

James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. "Scalable work stealing." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–11.

Ralf Hoffmann and Thomas Rauber. 2008. "Fine-grained task scheduling using adaptive data structures." In: *European Conference on Parallel Processing*. Springer, 253–262.

Maged M Michael, Martin T Vechev, and Vijay A Saraswat. 2009. "Idempotent work stealing." In: *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 45–54.

Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, and Jan F Prins. 2011. "Scheduling task parallelism on multi-socket multicore systems." In: *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, 49–56.

Junjie Qian, Witawas Srisa-an, Du Li, Hong Jiang, Sharad Seth, and Yaodong Yang. 2015. “Smartstealing: Analysis and optimization of work stealing in parallel garbage collection for java VM.” In: *Proceedings of the Principles and Practices of Programming on The Java Platform*, 170–181.

Arch Robison, Michael Voss, and Alexey Kukanov. 2008. “Optimization via reflection on work stealing in TBB.” In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–8.

Alexandros Tzannes, George C Caragea, Rajeev Barua, and Uzi Vishkin. 2010. “Lazy binary-splitting: a run-time adaptive work-stealing scheduler.” *ACM Sigplan Notices*, 45, 5, 179–190.

Lei Wang, Huimin Cui, Yuelu Duan, Fang Lu, Xiaobing Feng, and Pen-Chung Yew. 2010. “An adaptive task creation strategy for work-stealing scheduling.” In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 266–277.