Work Stealing Algorithms

1. Introduction

Parallel computing in a multiple instruction/multiple data system often involves the generation of a very large number of tasks. The distribution of these tasks among worker can have a significant effect on the performance, and typically makes use of some sort of dynamic load balancing. There are a number of algorithms to implement dynamic load balancing. One of these algorithms is work stealing, of which a number of variants exist. This report will mainly focus on work stealing algorithms and the factors involved in its performance.

There are a number of different work stealing approaches that have been proposed. The classical work stealing algorithm, proposed in a paper by Blumofe & Leiserson (1999) has an identifiable upper bound on memory usage, execution time and communication. However, there are still factors that can influence the performance. In order to identify these factors, a good understanding of the classical algorithm must be captured, which will be discussed in section 2 and 3.

Departures from the classical work stealing algorithm have also been proposed by other researchers in the field. This ranges from the use of different queue structures (such as circular (Chase & Lev, 2005) or split deques (Lifflander et al, 2012)), different ways to think about the stealing process (such as in Baseline or SuccessOnly work stealing (Kumar et al, 2016), or considerations of the locality of processors (such as in hierarchical work stealing algorithms). We will expand upon these in section 4.

This report also looks at Paratask, a Java transpiler that extends the language through additional keywords that declare parallelizable tasks. The current implementation of task distribution is highly modifiable, and currently supports both work-sharing, work-stealing, and hybrid distribution policies. ParaTask is examined in detail in section 5. This project also involves the implementation of a circular linked deque, the performance of which will be compared against the original ParaTask implementation. This will be expanded upon in section 6 and 7, along with an analysis of the performance compared to the original scheduling policies.

2. Classical Work Stealing Algorithm

The classical work stealing algorithm as described by Blumofe & Leiserson (1999) involves processors that each have their own deque of tasks. Processors can be in one of two states: either they are workers (they have tasks to work on) or they are thieves (there are no tasks in the deque and they have no tasks to work on). Workers continue to execute instructions in their task until a spawn occurs, at which point they will push the current task they are working on back to the bottom the deque, and operate on the newly spawned task. This ensures that instructions are operated on in the same order as a sequential implementation.

Processors that have no work are referred to as thieves, and look for other processors to steal tasks from. These thieves steal from the top of the deques of victim processors, effectively taking the oldest tasks in the deque. The effect of this is two-fold: thieves now have work to do, meaning that they are now workers, and the stolen tasks tend to be larger in granularity, meaning they will spawn further tasks for the thief.

The classical work stealing algorithm exhibits various design choices that may influence the performance of the work stealing algorithm. For example, what are the most effective data structures to facilitate work stealing? Which tasks should run on which processors following a spawning operation? And how many tasks should a thief steal in a single round of stealing? These are all questions that we will attempt to answer in this report.

3. Factors influencing Work Stealing and Work-Stealing Variants

A number of design choices, as mentioned earlier, can influence the performance of a work stealing algorithm. These choices are summarized below.

- The set of data structures being used - for example, in the classical algorithm, each processor has a deque of tasks, but the type of data structures and the organization of such structures is worth inspecting.
- The choice of task to execute following a spawn operation - in the classical algorithm, the newly spawned task is always executed on the same processor, leaving the parent available to be stolen. This is known as continuation stealing, or parent stealing (Robison, 2014).
- The victim selection process. The classical algorithm uses a random selection algorithm. This is provably efficient if we do not consider communication costs between processors, but there may be other methods to select victims.
- The number of tasks stolen. Clearly this should not surpass half the tasks from the victim's task stack (as this would promote suboptimal load balancing), but a choice must be made on how many tasks to steal. The classical work stealing algorithm steals a single task at a time.

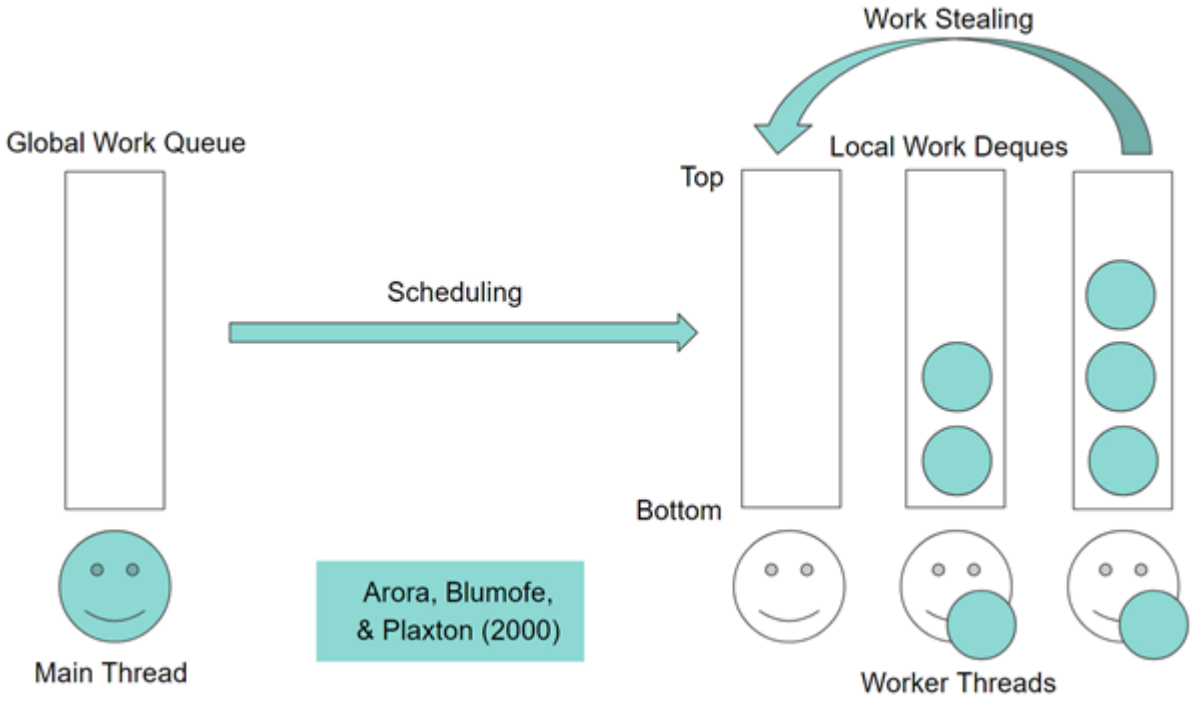
4. Work Stealing Algorithm Variants

Variants on the classical work stealing algorithm have been proposed in order to increase the performance of the algorithm. Typically, one or more of the aforementioned factors is changed in order to support a specific need of an environment running the algorithm. In this section, various work stealing algorithms will be examined, focusing on how they differ from the classical work stealing algorithm and what this means for performance.

a. Dynamic Queue Structures - Non-Blocking Dynamic Work Stealing Deques

This algorithm is based on the non-blocking work-stealing algorithm of Arora, Blumofe, and Plaxton (aka ABP work-stealing). It focuses on a specific queue structure, and aims to avoid overflow in the working queues.

In the original ABP work-stealing algorithm, a main thread and several worker threads are required to organize the environment. The main thread is responsible for dividing the original large task into subtasks, which is done before the execution of any other tasks and cannot be stolen. The main thread maintains a global work queue which can be accessed by all threads. Each worker thread also maintains a local work deque, which is an array-based double-ended queue (deque) that can be accessed from both top and bottom end. New tasks assigned by the main thread are pushed into the bottom end. The owning thread pops tasks

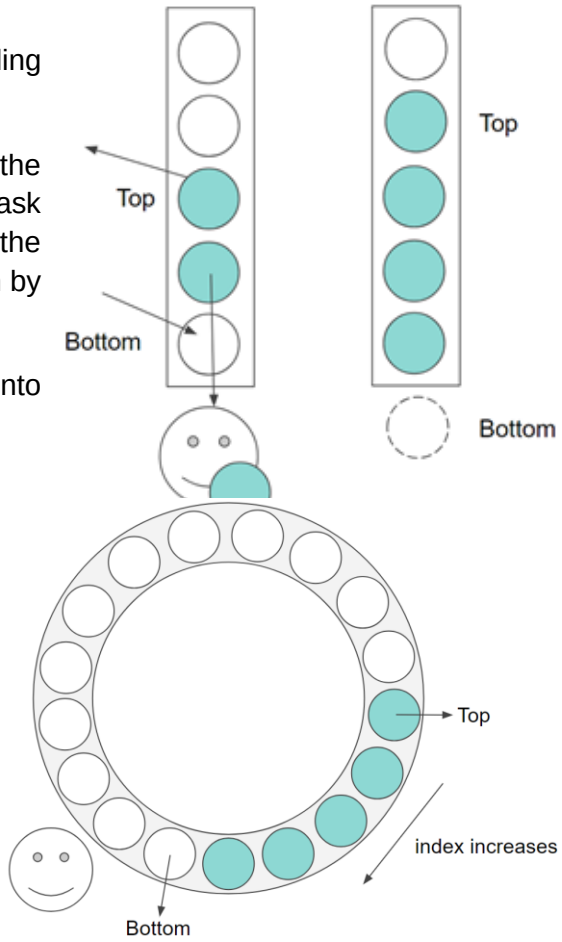


from the bottom end of the deque. Other worker threads steal tasks from the top end. No element can be pushed to the top end.

After a subtask is produced, it is initially put into the global work queue, and then assigned to a worker's local work deque. In this step, the main thread decides which worker thread to assign the subtask to, which is called scheduling. Work-stealing happens after the scheduling process. When a worker thread finishes its current task, it would look at its local work deque for a new task. If the local work deque is empty, then it turns to the global work queue to find if there is any new subtask produced. If the global work queue is also empty, then work-stealing happens – it will choose another worker thread, and try to steal a task from the top end of its work deque.

There are three main functions in this work-stealing algorithm:

- Task steal(): Returns Empty when the deque is empty. Otherwise, returns the task successfully stolen from the top of the deque, or returns null if this task is stolen by another worker thread.
- pushBottom(Task task): Pushes task onto the bottom of the deque.
- Task popBottom(): Returns Empty when the deque is empty. Otherwise, pops an task from the bottom of the deque.



In the figure above, the indicator "Top" indicates to the topmost task in the deque,

while “Bottom” indicates to the next available slot. Even if the deque is not full (there is still an available slot in the top end of the rightmost deque), the deque will overflow when a new task is pushed.

A circular deque is presented to fix the overflow problems, which is shown in the figure. In this structure, if the pushBottom() method is called, the deque grows automatically when the deque is becoming full after this push. In this case, the deque can be used more efficiently. Indicators move on the deque circularly, and each slot can be automatically reused, therefore there is no need to reset the slots to be empty after used. In addition, the circular deque gets full only when all the slots are occupied, and a new circular deque with size of current size plus one will be created and all elements in the current deque are copied to the new deque, so no overflow problem appears in this structure.

b. StealHalf Algorithm

The classical work stealing algorithm only steals one task at a time in its default implementation. Although this was sufficient for an optimal computation on the critical path of the program, multiple task steals were supposedly inefficient, as an atomic CAS operation (which can be costly) would be required for each push or pop to/from a deque. Hendler & Shavit (2002) have shown that it is possible to allow processes to steal up to half of a victim’s tasks that is non-blocking and minimizes the number of atomic CAS operations.

The StealHalf algorithm involves changing both the number of tasks that is stolen, as well as changing the queue structure in order accommodate the granularity of steal operations. StealHalf uses an extended deque structure for local deques. The additional properties of this circular deque are twofold; it is circular in nature, and it encapsulates an additional property (called stealRange in the literature) - this essentially defines the range of items that is available to be stolen. StealRange is updated in two conditions: either when the number of items surpasses or falls below a power of two, or when a successful steal operation has occured since the last time StealRange had been updated.

For any monotonic sequence of n pushes/pops (that is, for any sequence of pushes/pops where the counter only increases or only decreases), StealHalf only requires at most $\log n$ CAS operations, which is significantly better than the previous multi-steal algorithm (which had at most n CAS operations).

c. Hierarchical Work Stealing

Hierarchical Work Stealing (HWS) is a variant of the classical work stealing algorithm, which uses information about the locality of processors. Local processors typically have lower communication costs compared to remote processors - e.g. send tasks between processors in a single cluster will be less costly compared to sending tasks between clusters.

Hierarchical Work Stealing changes a number of factors - which tasks can be stolen, the victim targeting method, and the data structures involved. It does this by dividing the platform into clusters. Processing elements belong to a cluster if they are connected by relatively fast communicating network links. For example, processors on the same computer could be considered a cluster, meaning there would be one cluster for each machine in a multi-computer network.

Ordinary cluster elements may only send steal requests to other elements in the cluster - that is, they can only make local steal requests. However, each cluster has a designated leader node that can make remote steal requests. Leaders can only work on and steal larger tasks. The size, or level, of a task is determine by the number of spawns required to create the task - for instance, the root task (the initial task) has a level of zero, but a direct descendant of the root task will have a level of one. These large tasks are also called global tasks, as they can be stolen by all other leaders globally. The number of global tasks that are present at any time is limited to a maximum of a number chosen by the user (typically it is a small number).

Each leader has two deques that can be stolen from. One deque is for global tasks, and can only be stolen from by other leader nodes in other clusters. The other deque is for tasks smaller than the level limit, called local tasks. Tasks in this deque can be stolen by nodes belonging to the leader's cluster.

Normal cluster elements (also called follower nodes or slave nodes) send steal requests either to each other or to the leader node. Only local tasks can be stolen by follower nodes from cluster leader. Leader nodes supply their cluster with additional tasks by working on global tasks and breaking them down into subtasks, which may be small enough to be stolen by local nodes.

Hierarchical work stealing has been observed to outperform the classical work stealing algorithm in practice, as the average steal latency decreases. They have also been analyzed theoretically by Quintin & Wagner (2010). They found that the number of global steal attempts is bounded (suggesting good load balancing between clusters), and a speed up of around 20% compared to classical work stealing algorithms on a distributed platform.

5. ParaTask

ParaTask provides a few different scheduling and distribution options. This can be set by a developer using ParaTask.setScheduling() method, which takes a ScheduleType enum as a parameter. The types of scheduling that can be set are as follows:

- ScheduleType.WorkSharing - this is a work sharing policy, where all tasks are sent to a global queue, and then distributed to each processor in a FIFO policy.
- ScheduleType.WorkStealing - this is a work stealing policy, identical to the ABP classical work stealing algorithm. There is no global queue and each processor is equally likely to steal from any other processor.
- ScheduleType.MixedSchedule - this is a combination of both policies, where each processor has a local deque of tasks that can be stolen from, and a global task queue also exists. Enqueuing tasks to local queues involves a work stealing policy if the queue belongs to a worker thread, and a work sharing policy otherwise.

ParaTask also provides some additional variants on work stealing scheduling, all of which limit the amount of tasks that can be present on the system at any one time. These are as follows:

- WorkFirstTaskDepth - if a task is of a certain depth or level, it must be processed. This has similarities to how HWS divides its tasks into global or local tasks.

- WorkFirstLocal - the number of tasks in a local queue cannot exceed a particular number - tasks spawned past this must be directly processed.
- WorkFirstGlobal - the number of tasks in the entire system cannot exceed a particular threshold. Similar to the above, tasks spawned past this point must be directly processed.

6. Implementation

Currently, in the ParaTask project, the work-stealing queue is implemented in the following way:

- Each worker thread maintains a local work queue of “LinkedBlockingDeque<E>” structure. New elements are added to and removed from the head (indicated by “first” in the LinkedBlockingDeque structure) of the queue using a last in first out (LIFO) policy, and are stolen by other worker thread from the tail (indicated by “last” in the LinkedBlockingDeque structure) using a first in first out (FIFO) policy.
- LIFO end: elements are (1) assigned from the global queue to the local work queues, and (2) popped by the work queues’ owner threads, from the same end of the it. In this end, the earlier the elements are assigned, the deeper they are stored, and the latter they are accessed by the owner thread.
- FIFO end: elements are stolen by other threads from the opposite end of the one they are added. In the stolen end, the earlier the elements are added, the earlier they are accessed by the thief thread.

There are two closely related classes: WorkStealingQueue and FifoLifoQueue. In WorkStealingQueue, the local queues for all threads are stored in a ConcurrentHashMap structure. This hashmap is indexed via the corresponding owner thread id. FifoLifoQueue extends WorkStealingQueue and defines a LinkedBlockingQueue structure, which is used as a global queue.

We introduce a new class called CircularBlockingDeque. This class extends LinkedBlockingDeque, replacing WorkStealingQueues as the default implementation for local queues. In this structure, the next node of the last node is set to the first node, and the previous node of the first node is set to the last node. It can thus dynamically resize, meaning the capacity of local work queues can be set to a small number instead of the max value of integer, saving memory. The corresponding iterator is also rewritten to accommodate these changes, and ensure that hasNext() returns the correct item.

Another new class named CircularWorkStealingQueue is created by modifying the existing WorkStealingQueue class. All the local work queues are defined in CircularBlockingDeque structure in the new class. In addition, the FifiLifoQueue is now defined as a subclass of CircularWorkStealingQueue. Related work queue structures are also modified.

Although the Non-Blocking Dynamic Work Stealing Deques uses atomic operations such as the Compare And Swap operation for thread safety, locks are still used in this implementation. The main reason is the time limit, as the original project uses linked blocking

deques, it is better to make changes based on what we have, otherwise there would be several risks.

7. Experimental Results

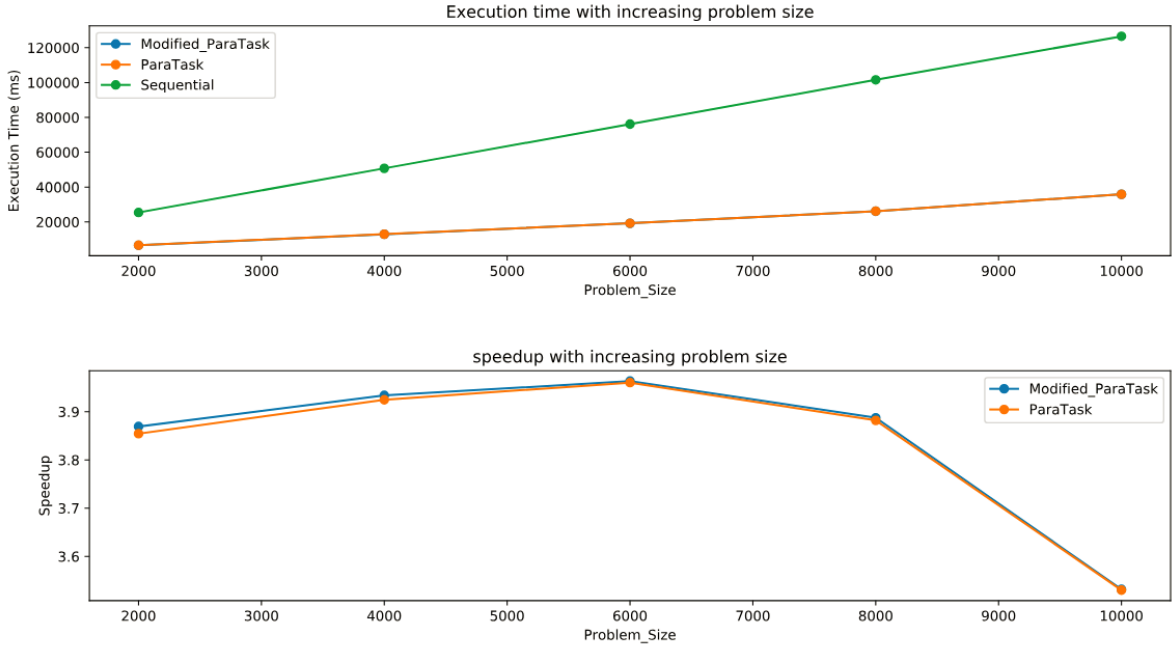
The motivation of the benchmark project is to compare the performance of our work-stealing algorithm which is integrated in ParaTask runtime to the built-in work-stealing algorithm of ParaTask. A simple compute-intensive benchmark is introduced in this project where both problem size and granularity are controllable in order that both work-stealing algorithms are running in the same condition.

The benchmark project is running on the lab computer that the OS is Linux (Ubuntu v16.04.4 LTS) with 4 physical x86_64 based processors.

The benchmark starts with work-stealing schedule type setting in ParaTask so that the work stealing mechanism is being used. To compare the performance of these two algorithms, we introduced two approaches which are:

- Compare the execution time of sequential, ParaTask and our implementation with increasing problem size
- Compare the speedup of ParaTask and our implementation with increasing problem size

The plotting graphs is shown below:



As we can see that the sequential execution time is increasing linearly with the increasing problem size. This is because the work item is balanced so that two algorithms are running the same workload. The graphs show that the two work stealing algorithms have very similar performance, but we can still see a little bit improvement in our work-stealing implementation. And the speedup of both algorithms falls down after the problem size reaches 6000.

References

Blumofe, R. D., & Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, *46*(5), 720-748.

Chase, D., & Lev, Y. (2005, July). Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures* (pp. 21-28). ACM.

Hendler, D., & Shavit, N. (2002, July). Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (pp. 280-289). ACM.

Kumar, V., Murthy, K., Sarkar, V., & Zheng, Y. (2016, November). Optimized distributed work-stealing. In *Irregular Applications: Architecture and Algorithms (IA3), Workshop on* (pp. 74-77). IEEE.

Lifflander, J., Krishnamoorthy, S., & Kale, L. V. (2012, June). Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing* (pp. 137-148). ACM

Robison, A. (2014). *A primer on scheduling fork-join parallelism with work stealing*. Tech. Rep. ISO/IEC JTC 1/SC 22/WG 21, The C++ Standards Committee.

Quintin, J. N., & Wagner, F. (2010, August). Hierarchical work-stealing. In *European Conference on Parallel Processing* (pp. 217-229). Springer, Berlin, Heidelberg.

Contributions

Group22 Members	**Contributions**
GuoLong Kang	Implementation of benchmarking Section 7 of report
Michael Ieti	Minor test input Section 1-3, 4b-5 of report
Sha Luo	Implementation of the circular blocking deque. 4.a Dynamic Queue Structures - Non-Blocking Dynamic Work Stealing Deques and Implementation part of the report.