SCEEM: Stealing and Chunking Execution Engine for Mesos

Alex Degtiar and Apoorva Sachdev

Introduction

The primary problem we pursued for this project is to address the latency of executing many small tasks in a heterogeneous cluster environment. Many distributed frameworks execute tasks one at a time per slave node from a centralized scheduler. This is acceptable for medium-term and long-term tasks, where the amount of work done for that task outweighs any latency overhead for sending the task to the slave node on which it is executed. However, the smaller a task becomes, the more such overhead dominates the end-to-end cost for task execution. This is certainly the case on Mesos[1], a cluster manager that provides resource sharing among various distributed frameworks and, more importantly in our case, serves as a platform on which such frameworks can be implemented.

Mesos provides a framework developer's API to receive heterogeneous resource offers (e.g. CPUs and memory), and allows you to launch tasks to run on the corresponding slaves. This has the same limitation as in the frameworks previously mentioned: the total execution time of small tasks launched through this interface is dominated by the latency overhead of shipping the task to the slave. To solve this limitation, we chose a two-part solution. First, we implemented an extension to the Mesos API which lets you submit many tasks in one chunk. This chunk can be of independent tasks, or the dependencies can be specified. This negates the overhead of launching each task individually, but has the downside that insufficiently balanced task chunks can cause significant idle time for the chunk that finishes first. We then extended Mesos with a task-stealing scheduler module to provide a dynamic load balancing solution to this issue. Contrary to most task-stealing schedulers, such as that of Cilk[2], task stealing on Mesos needs to take into account the heterogeneous resources within the cluster. In this report, we explore how employing a chunking and stealing scheduler such as SCEEM can significantly improve the aggregate performance of executing many small tasks in system like Mesos.

Overview

SCEEM is an abstract framework built on top of the Mesos platform. SCEEM extends the Mesos API in two ways by implementing task chunking and task stealing. SCEEM provides abstractions to the underlying user-defined framework to group small tasks as task chunks. It also provides an abstract task-stealing scheduler, which decides when to steal and selects which tasks should be stolen based on certain resource constraints. The underlying framework can use this abstract task-stealing scheduler to take advantage of task-stealing without implementing it.

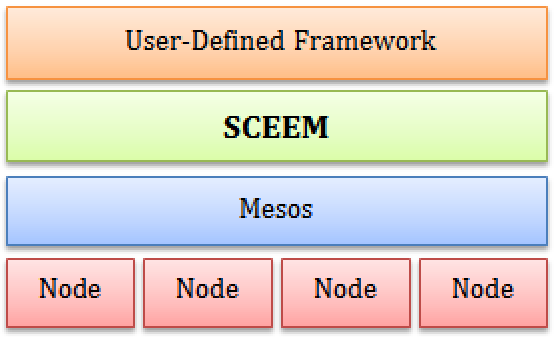

Figure 1: Overview of where SCEEM fits in the Mesos platform

The SCEEM scheduler still relies on the underlying scheduler to do the actual scheduling and forming well-sized tasks for each offer that it receives. However, it hides the complexity of dealing with task chunks at the Mesos level. It enables the underlying framework to minimize the cost and latency associated with launching tasks by launching task chunks instead of multiple small tasks.

Task stealing is implemented as a abstract-scheduler on the existent MESOS platform. Depending on the resources available to the scheduler and the tasks currently running, the Task-stealing scheduler decides whether it wants to reduce the load of another node by stealing some of the tasks from it. The actual process of stealing is black-boxed for the underlying framework and a default implementation of an on-line, centralized, greedy task-stealing algorithm is provided. However, the underlying framework can configure when to steal and which tasks to steal by overriding the default implementation.

Implementation

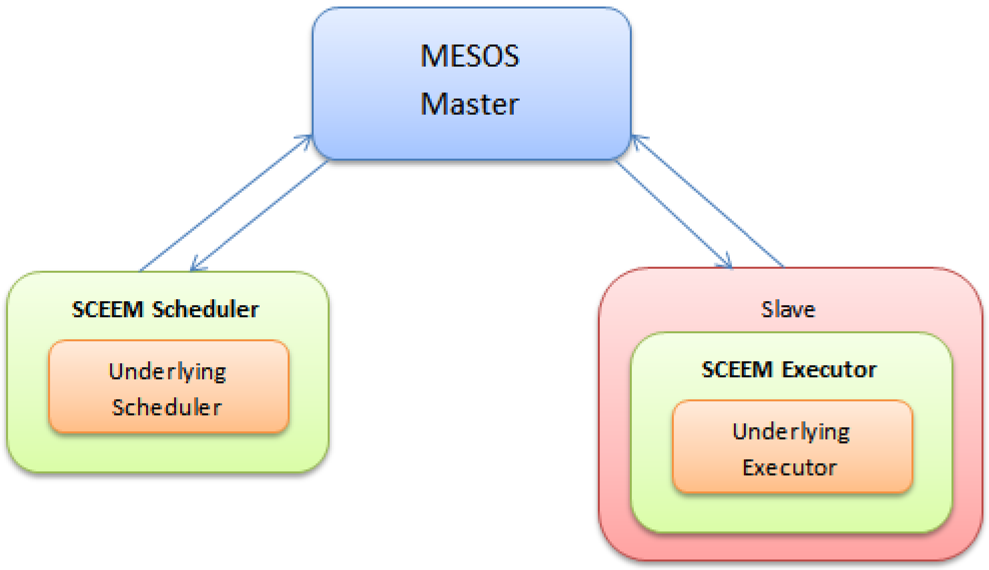

Figure 2: Detailed diagram of SCEEM and Mesos Architecture

Task Chunking

Task chunking can be split into two main components, the task-chunking scheduler and the task-chunking executor, where the task-chunking scheduler forms a wrapper around the underlying framework scheduler and the task-chunking executor forms a wrapper around the underlying framework executor.

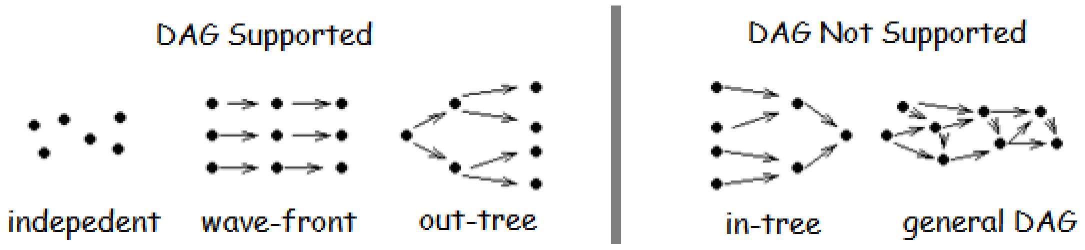

Figure 3: DAG structures

The task-chunking scheduler enables the underlying scheduler to form and launch task chunks. A task chunk is a directed acyclic graph (DAG) or queue of pending tasks. We currently only support the independent, wave-front, and out-tree DAG structures but our framework could be easily extended to support more varied DAG structures. To add a dependency between two tasks, we can add one task as the subtask of another which will enforce that the first task is executed before its subtask. The task-chunking scheduler controls the launching of these task chunks. Since, the Mesos platform is currently unaware of tasks existing within task chunks, updated related to individual tasks are sent as framework messages. These framework messages are then parsed by the task-chunking scheduler and are replayed to the underlying scheduler. The task-chunking scheduler also provides an interface for the underlying scheduler to kill either a running task chunk or certain subtasks within a task chunk.

The task-chunking executor enables task chunks to be run on the underlying executor. It uses a task table to maintain an internal queue of pending tasks. Once it receives a task chunk from the mesos master, it adds the task chunk and all its subtasks to the task table with a ‘pending’ status. The task-chunking executor uses an event-driven loop to run pending tasks on the underlying executor one at a time. It receives status updates pertaining to the tasks from the underlying executor, updates the pending task table, and relays the status updates back to the task-chunking scheduler. The task-chunking executor extends the Mesos API by supporting the killing of task chunks and killing of subtasks within a task chunk. In order to kill a task chunk, it removes the task chunk and recursively removes all of its subtasks from the task table of pending tasks. If any of the subtasks are currently running on the underlying executor, it invokes the kill task method of the underlying executor. In a similar way, in order to kill certain subtasks from a task chunk, those subtasks and their children are removed from the pending task table and if a subtask is currently running, the underlying executor’s kill task method is invoked.

Task Stealing

Task stealing can be implemented on the Mesos platform by extending the task-chunking scheduler and using the task-chunking executor. We extend the task-chunking scheduler by adding a task table similar to the pending task table in the task-chunking executor. The task table maintains the state of task chunks and tasks currently running. Once the task-stealing scheduler receives an offer, depending on the order specified by the underlying scheduler, it decides to steal tasks or pass the offers to the underlying scheduler. If it decides to steal, it uses the default task-stealing algorithm or a user specified algorithm to select the tasks to steal from the task table. After selecting tasks to steal, it requests the respective executors to kill the stolen tasks, so at any time, one task is assigned to only one executor. It then creates task chunks with the stolen tasks as subtasks, and launches them with the new offers. If it decides to pass the offer to the underlying scheduler, the underlying scheduler returns a list of task/task chunks to launch.

Task-stealing algorithms

We implemented a general, straightforward, greedy algorithm for performing the default task stealing. The algorithm is invoked when Mesos forwards a list of offers, each corresponding to free resources available on a slave, to the task-stealing abstract scheduler. If a single offer is received, the single highest-priority task chunk currently running on any slave which can run within the offered resources is selected as a donor for stealing. If the highest-priority task chunk cannot run within the offered resources, the next-highest priority task chunk is chosen. Priority is configurable, but is currently defined as the task chunk with the biggest tree (or portion) of idle subtasks, as to pick the largest idle portion. The selected task chunk is split, half of its subtasks are stolen into a new task chunk to launch, and its priority is re-evaluated. Any remaining portion of the offer resources is sent through the algorithm again. If more than one offer is initially received, then the offers are processed in a queue prioritized by smallest resource amount (first by cpu, then by memory, then by anything else). That way, larger offers (such as for nodes with many cpus) are not split unnecessarily when that many cores may be needed by a slightly lower priority task chunk that needs more resources. This may not be perfect, but the multidimensional nature of the resources makes it more difficult to make ideal decisions with splitting. Any unused portion of the offers after going through this stealing process is then forwarded to the underlying framework for typical consumption and task launching. This task stealing selection process is also overridable by the underlying framework, as is the ordering of the stealing and the underlying offering.

Alternative algorithms we explored mostly involved different prioritizations to determine the selection of a donor task chunk. A problem with using the number of subtasks in the chunk as a means to select a donor is that we cannot know in advance how long each subtask actually takes to run. An alternative priority we considered was the chunk with the longest-waiting idle subtask. That way, small-sized chunks with long-running subtasks, some of which could be waiting for a really long time, can be selected ahead of large chunks with trivially-sized subtasks, which the former prioritization would always pick first. Also, the prioritization of the chunks can be further influenced by execution data we could collect as tasks are launched and completed. That way, we might be able to link task descriptions with prior execution times to estimate the real task chunk workload. Such an estimate can also be user-configured, such as specifying whether a task is small, medium, or large. We also considered other algorithms besides greedy donor selection + splitting. We chose single-task-chunk splitting in order to maintain as much as possible potential locality constraints for the subtasks. This is based on an assumption that tasks within a chunk, whether independent or within a DAG of some sort, may have locality constraints that make them faster execute together on the original node they are assigned to. It is preferred, then, to keep subtasks together as much as possible. For cases that this assumption does not hold, we could have implemented an algorithm that more evenly steals subtasks from all appropriate task chunks, and puts them together into a new chunk to launch. We could also have prioritized pairing where the donor task chunk and the offer correspond to the same slave node. We could even combine some of these approaches under a weighted score priority approach. We also considered treating this problem as a variant of the knapsack problem, the solution to which may be more ideal than our greedy approach. Other improvements we can make are in the handling of task dependencies.

Due to context of SCEEM as an abstract scheduler, the algorithm chosen is in regard to dynamic scheduling done with very partial information passed to us from the underlying scheduler. The task-chunking scheduler is passed an in-memory partial DAG of a potentially much-larger DAG, and key task size information is not available until the task is complete (unless some is provided by the underlying scheduler). The chunk size can be highly variable, determined appropriately (or inappropriately) by the underlying scheduler. The stealing and run-time task behavior ends up being somewhat similar to guided self-scheduling, since large task chunks are initially submitted, and then smaller and smaller tasks are run if stolen. This is determined dynamically by the algorithm, however, instead of having to actually make task chunk size decisions. The stealing is effective for both high and low variance tasks. If the variance ends up being high, the stealing will take care of it by re-distributing tasks. In a way, our task stealing scheduler ends up performing a variant of weighted factoring. Due to the heterogeneous nature of cluster resources, resource offers vary in size. We split offers into pieces based on how much resources the stolen tasks require, which ends up running more stolen tasks on compute nodes that have more resources than others.

Results

In order to benchmark the performance of SCEEM, we model and simulate the execution of parallel tasks represented by directed acyclic graphs, each on a system of 3 slaves (distributed nodes). We compared the regular Mesos implementation with the SCEEM task chunking implementation to analyse the benefit of task chunking vs launching multiple small tasks. Our tasks were all independent. Since we were running the slaves on the same machine as the master and scheduler, the latency of communication was not significant. Thus, we simulated latency by adding (20 milliseconds * num of times the slave sent an offer of resources to the scheduler) to the total time taken. This is reasonable since every-time a slave sends an offer to the scheduler, it is in idle state and has no tasks to run while a task is being sent over to the slave for execution.

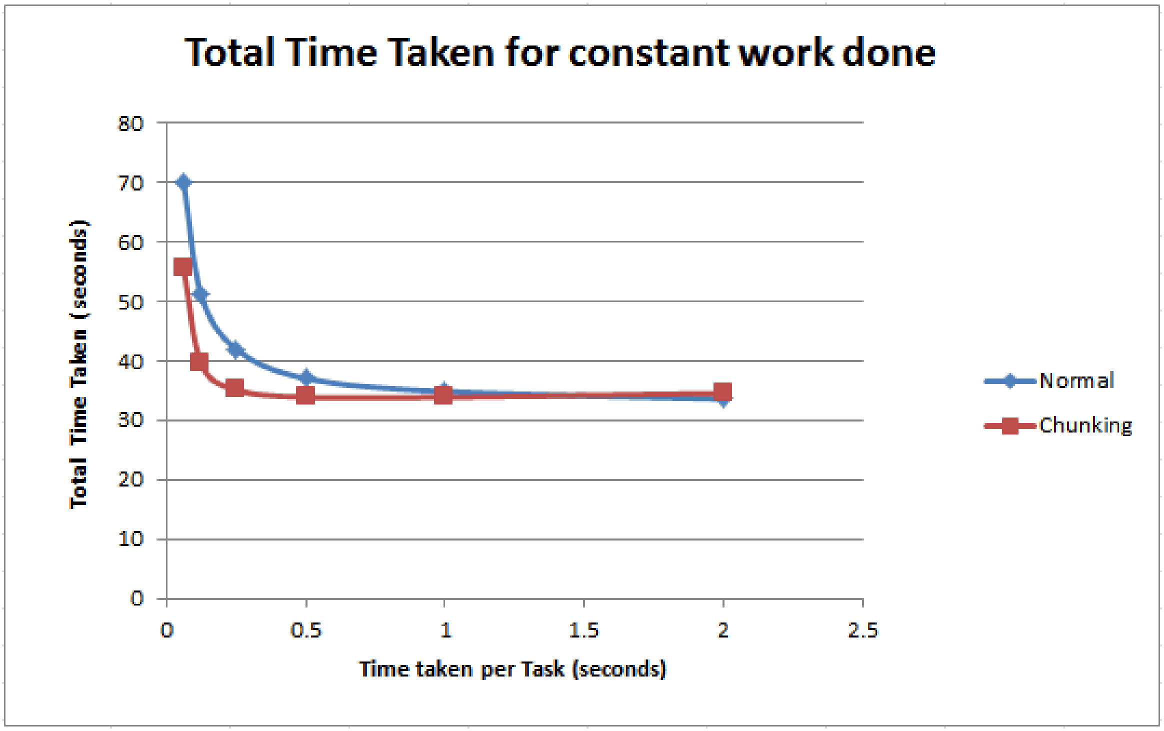

Figure 4

The graph above shows how the total time taken varies with time taken per task for constant work done. (As time taken per task decreases the number of tasks increase). It can be seen from the graph, when the time taken per task is low, SCEEM task-chunking framework performs better than the regular mesos framework. This is intuitive, as for smaller tasks, the latency overhead is significant whereas for larger tasks, the latency overhead becomes less significant and hence it may not be as worthwhile to do task chunking. Also, the difference between the total task time shown in the graph is an underestimate because the regular Mesos uses a native

C++ implementation for a scheduler and executor which runs much faster than our python implementation. This python overhead is pretty significant, because ignoring latency we can see a noticeable difference in the total time taken to run the test between the normal and the chunking implementation for the same value of time taken per task.

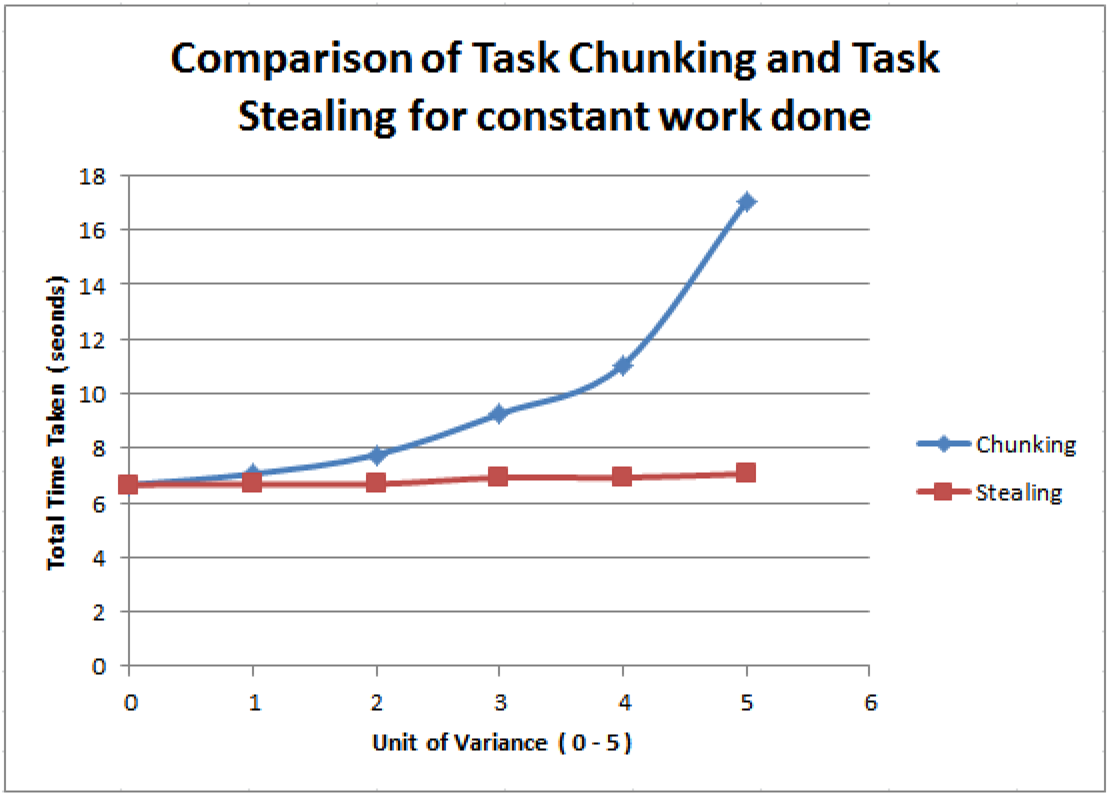

Figure 5

We also gathered results for the effectiveness of regular task chunking vs. task chunking with stealing. We compared their relative effectiveness as a function of how much the task chunk size varied. We used normally distributed task chunk sizes generated with different sigmas, but for simplicity, we've graphed this above on a scale of 0-5, 0 being no variance (all chunks the same size), and 5 being a large amount of variance (size difference between smallest and biggest chunk is huge). As expected, we find that with the same the task chunk sizes, there is effectively no difference between stealing and no stealing. However, as the task chunk sizes vary more and more, it is clear that task stealing can dynamically balance the varied load with a minimal latency/stealing overhead.

Conclusions and Future Work

We employed SCEEM, a task chunking and stealing abstract scheduler on the Mesos platform and evaluated its performance. From the results, we can see that chunking tasks can significantly reduce the latency overhead especially for small tasks as the task-chunking scheduler performs better than the Mesos scheduler. For small tasks launched through the Mesos interface the total execution time is dominated by the latency overhead of shipping the task to the slave, thus by using the task chunking scheduler we can reduce this latency cost by a significant amount, however this becomes less significant as time per task increases. Furthermore, comparing task chunking and task stealing for varying task chunk sizes clearly improves cluster utilization, and subsequently the significantly reduces the total computation time for all tasks. We have shown that employing a chunking and stealing scheduler such as SCEEM can significantly improve the aggregate performance of executing many small tasks in system like Mesos. For future work, we can improve the task stealing algorithm, implement SCEEM natively, implement a distributed work stealing model instead of a centralized scheduler, and add full DAG support.

References

1. Mesos - http://incubator.apache.org/mesos/
2. Cilk - http://supertech.csail.mit.edu/cilk/
3. CIEL - http://www.cl.cam.ac.uk/research/srg/netos/ciel/