Weasel

October 6, 2015

1 Introduction

An important part of large-scale scientific applications is represented by many-task computing (MTC) workflows, found in a wide range of scientific domains, including astronomy, bioinformatics, data analytics. These applications are composed of a large number of tasks, e.g., from thousands to milions, which do not require strict coordination of processes at job launch, but instead, they exhibit inter-task dependencies expressed by means of files. The properties of MTC tasks lead to interesting scheduling opportunities: not only they have small execution times, e.g., a few seconds or minutes, but they also have fine-grained resource requirements, e.g., they use less than 1 CPU core, which are also varied between task themselves, e.g., some tasks require more CPU while others are I/O intensive.

Many efforts were made in designing scheduling solutions, capable of scale horizontally to large numbers of nodes and running tasks. These solutions focus on the decentralization of resource management and efficient in-memory data storage mechanisms. Nevertheless, improving the resource utilization of the node was seldom considered. The main mechanism to run tasks per node is to split the node's capacity in a fixed number of slots, e.g., 1 CPU core and 1 GB of RAM, and to assign for each task one slot. This static allocation leads to poor utilization, when tasks might not use all their allocated slot resources, or poor performance, when tasks require more than their allocation.

Another approach is to consider that task resource utilizations are known and the resource capacity can be fully allocated among tasks based on this information. However, this requires full information about all possible resources used by a task, which can prove difficult to obtain, e.g., priviledged access, as there are cases in which coarse-grain information, e.g., CPU, network bandwith utilization, is not enough. For example, large scale applications might access data from a remote file system shared with other applications. In this case, data needs to be transfered over the network to the node on which the task is running, and measuring the resource utilization of the task returns results regarding the *network bandwith utilization* of the task, among others. Thus, if the utilization for other resources is smaller, one might compute a number of tasks per node to saturate the network bandwith. However, this yields a wrong result as the number of tasks to run should be computed with regards to the capacity of the remote file system. When the remote file system becomes a bottleneck, e.g., the disk is slow

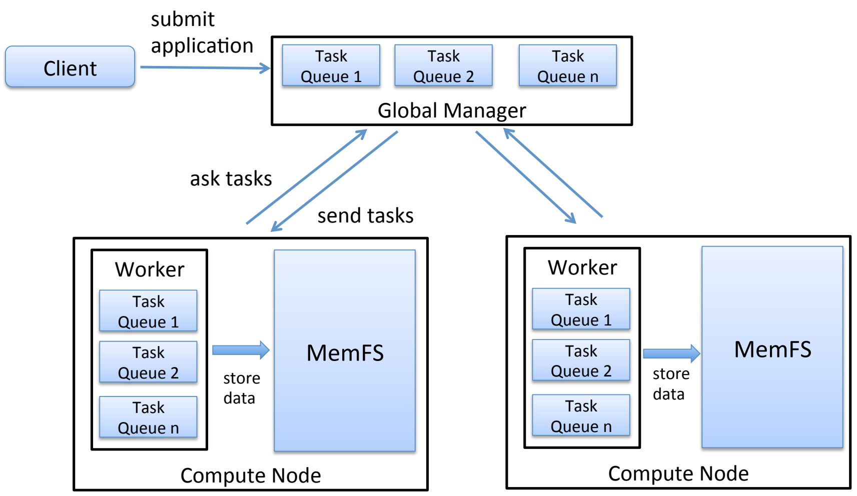

Figure 1: Scheduler architecture.

and/or other applications access large data amounts from it, running more tasks on the node might lead to worse application performance or it might degrade the performance of the other applications accessing data from the shared file system.

In this paper we present a scheduler for many-task computing applications that maximizes the resource utilization of the infrastructure by co-locating dynamically tasks on nodes, i.e., it vertically scales the application, with respect to resource utilization bottlenecks. The scheduler is composed of distributed independent workers which, during the application execution, learn the resource utilization characteristics of tasks and decide at runtime how many tasks to run and of each type to maximize the node resource utilization while avoiding resource contention. With each worker adapting locally, the scheduler adapts the overall application demand to fluctuations in resource availability in a friendly manner, allowing other applications which share the infrastructure to achieve good performance.

2 System's Architecture

The architecture of Weasel is shown in Figure 1. To improve the application's performance, the scheduler stores the application generated data in a distributed in-memory file system, i.e., MemFS. This aleviates the throughput limitations imposed by the disk and transforms the I/O intensive tasks in network intensive.

The scheduler is composed of a Global Manager and a set of Workers. An user submits an application to the Global Manager which creates a set of tasks queues, based on the application composition: there is one task queue per task executable. The task queues can be either *ready* or *waiting*. First, tasks are queued in the *waiting* queues and when their input data becomes available they are moved in the *ready* queues. To run tasks, each worker polls the Global Manager. Each worker decides the number and type of tasks, i.e., from which queue, to run by using the vertical scaling policy.

Workers are only knowledgeable of the tasks from the *ready* queues of the Global Manager as tasks from *waiting* queues cannot be executed. When a *ready* queue becomes populated with tasks, the Global Manager informs the workers. When the worker receives a task type that wasn't previously encountered, it goes in a profiling mode. In this mode, the worker will run one task per node and register its resource utilization several times.

After a worker learns the task resource utilization it creates a local queue specific for the task type. When tasks finish their execution, the worker asks the Global Manager for more tasks of the specific type and stores them in the local queue. The local queue is needed because, to avoid idle periods, which can influence the resource utilization measurements, the number of asked tasks is higher than the number of finished tasks.

Measuring the Resource Utilization Each worker measures the resource utilization of its compute node and keeps its history. This information is used by the vertical scaling policy, as further described in Section 3 The resource utilization is read at small time intervals, e.g., 0.5 seconds, to better capture the resource contention, and the values are aggregated as follows: a histogram is computed to determine the interval in which most of the values belong and then the average of values from this interval is returned.

There are two reasons for this strategy. First, the task's resource utilization is not constant during its runtime and we want to capture exactly these contention times. For example, a task might read a file, perform some computation and write the results in another file. Second, we need to smooth the read values while keeping a measuring interval small enough to allow the worker to adapt fast the number of tasks.

To further smooth the resource utilization series, each worker keeps a table that corelates resource utilization to the number of tasks running on the node. When a new value is read, it is added to the previous value that coresponds to the current number of tasks, using an exponential moving average function.

3 Worker's Scaling Policy

Workers dynamically co-locate tasks on nodes during application runtime based on their resource utilization characteristics. Each worker applies the following periodically: at each time interval it detects if resource contention appears and adapts the number of tasks accordingly. The adaptation policy is used to share the node between tasks which belong to the same application, but are in different task queues and have compatible resource utilization characteristics. We assume that tasks from the same queue, i.e., which have the same executable, have the same resource utilization.

A worker decides to co-locate tasks from different queues if they do not have the same dominant resource, i.e., the resource for which the tasks have a maximum utilization. If different queues cannot be co-located, their tasks are processed in a first-come-first-served manner. For co-located queues, the worker applies the adaptation policy

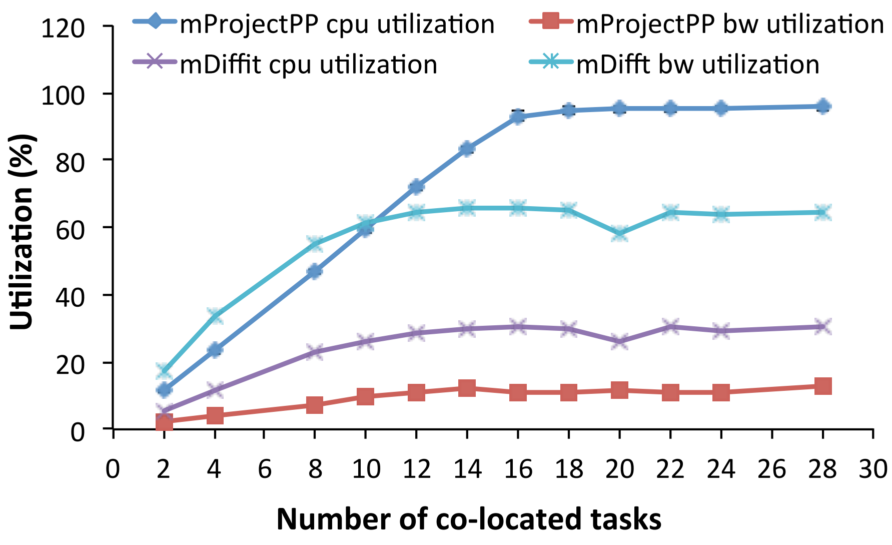

Figure 2: Resource utilization for different tasks beloging to an instance of Montage.

in parallel; for each queue, the policy tracks the contention for the task's dominant resource.

3.1 Contention Detection

To notice when contention appears, we ran a scientific application, Montage, on a compute node with 8 cores (16 with hyperthreading). Montage is a scientific workflow, composed of different stages in which different executables run. Some stages are parallel, i.e., they are composed of a large number of tasks, while others are serial. We stored the application's data in an in-memory file system, mounted on a separate set of nodes, and we varied the number of concurent tasks running on node.

Figure 2 shows the resource utilization of the node for the first two stages of Montage, called mProjectPP and mDifffit. We chose the first two as they have the most significant number of parallel tasks. We measured the cpu and network bandwith utilization, as tasks are CPU or I/O intensive. We notice that, while for mProjectPP the CPU utilization almost reaches the maximum CPU capacity of the node, the mDiffit stage does not manage to saturate the network bandwith, which remains at 60% utilization with increasing numbers of co-located tasks.

Using the previous observations, contention is detected in two cases: (i) when the absolute value of the change in the total resource utilization drops below a certain threshold; (ii) when the total resource utilization drops when increasing the number of tasks, i.e., the current total resource utilization is smaller than the maximum enountered so far and the current number of tasks is smaller than the number of tasks associated to the maximum. The contention detection is applied when changing the number of tasks.

Algorithm 1 Number of Tasks Adaptation

```
1: $past\_history = \emptyset$ {past ntasks values}
2: $good\_points = \emptyset$ {values of ntasks from which the number of tasks was increased, ordered decreasingly}
3: $bad\_points = \emptyset$ {values of ntasks from which the number of tasks was decreased}
4: $same\_dir = 0$ {used to control the agresivity of the policy}
5: $tasks\_increase = 0$
6:
7: **TaskAdaptation**($ntasks_{current}$, $resource_{diff}$, $T_c$, $start\_after$)
8: direction = 0
9: $ntasks_{old}$ = get last ntasks from $past\_history$
10: **if** $ntasks_{current} - ntasks_{old} > 0$ **then**
11:     direction = 1
12: **else**
13:     direction = -1
14: **if** $direction = 0$ **then**
15:     return $ntasks_{current} + 1$
16: **if** contention detected **then**
17:     $new\_direction = -1$
18:     $good\_points = good\_points - \{ntasks_{current}\}$
19:     $bad\_points = bad\_points \bigcup \{ntasks_{current}\}$
20: **else**
21:     $new\_direction = 1$
22:     $good\_points = good\_points \bigcup \{ntasks_{current}\}$
23:     $bad\_points = bad\_points - \{ntasks_{current}\}$
24: **if** $new\_direction \neq direction$ **then**
25:     $same\_dir = 0$
26:     $tasks\_increase = 1$
27: **else**
28:     $same\_dir = same\_dir + 1$
29: **if** $same\_dir > start\_after$ **then**
30:     $tasks\_increase = tasks\_increase \cdot \frac{|resource\_diff|}{T_{change}}$
31: **if** $new\_direction = 1$ **then**
32:     IncreaseTasks($ntasks_{current}$)
33: **else if** $new\_direction = -1$ **then**
34:     DecreaseTasks($ntasks_{current}$)
35: $past\_history[ntasks_{new}] = ntasks_{new} - ntasks_{current}$
36:
37: return $ntasks_{new}$
```

3.2 Number of Tasks Adaptation

Each worker adapts the number of tasks to maximize the resource utilization of the node without creating contention for resources. The number of tasks is changed in two phases: conservative and aggressive. In the conservative phase, the number of tasks is changed incrementally, while in the aggresive phase the number is changed proportionally to the change in resource utilization. The number of tasks is increased until contention is detected. Then, it is decreased until the resource utilization drops

Algorithm 2 Increasing Adaptation of Number of Tasks

```
1: IncreaseTasks($ntasks_{current}$)
2: exp = 1
3: $ntasks_{new} = ntasks_{current} + tasks\_increase$
4: while $ntasks_{new} \in past\_history$ and $ntasks_{new} > ntasks_{current} + 1$ do
5:     if $ntasks_{new} \in bad\_points$ then
6:         $ntasks_{new} = ntasks_{current} + max(1, \frac{tasks\_increase}{2^{exp}})$
7:         $exp = exp + 1$
8:     else
9:         break
10: return $ntasks_{new}$
```

Algorithm 3 Decreasing Adaptation of Number of Tasks

```
1: DecreaseTasks($ntasks_{current}$)
2: if $same\_dir$ = 0 then
3:     $ntasks_{old}$ = get last ntasks from $past\_history$
4:     return $ntasks_{current} - max(1, \frac{ntasks_{current} - ntasks_{old}}{2})$
5: $ntasks_{init}$ = get first value from $good\_points$
6: while $ntasks_{init} \geq ntasks_{current}$ and $good\_points \neq \emptyset$ do
7:     $ntasks_{init}$ = get next value from $good\_points$
8: if $same\_dir > start\_after$ then
9:     $ntasks_{new} = max(1, ntasks_{current} - max(1, \frac{ntasks_{current} - ntasks_{init}}{2}))$
10: else
11:     $ntasks_{new} = max(ntasks_{init}, ntasks_{new} - tasks\_increase)$
12: return $ntasks_{new}$
```

lower than the contention value. A change history is used to help the convergence to the contention value.

Algorithm 1 describes the policy. The algorithm receives as input the following information: (i) the current number of tasks $ntasks_{current}$; (ii) the difference in resource utilization between the current measurement and the previous one, $resource_{diff}$; (iii) a constant that is used to detect the resource contention, Tc; (iv) and a parameter used in changing the number of tasks, $start\_after$. The algorithm outputs the new number of tasks. During the application runtime, the algorithm also records all the changes it performs in a *change history*. This history contains the used numbers of co-located tasks, $past\_history$, the numbers of co-located tasks for which contention was not detected, $good\_points$ and for which contention was detected, bad_points.

In the first phase, the algorithm decides if it needs to increase or decrease the number of tasks (Lines 9-13). The algorithm uses a conservative phase at a change in the operation type performed on the number of tasks, e.g., the number was decreased previously and the algorithm decides that currently it needs to increase it (Lines 24-28). The transition between the conservative and aggressive phases is given by the parameter $start\_after$ (Lines 29-30), i.e., the highest the value of the parameter is the more conservative the algorithm is.

Task number increase Algorithm 2 describes how the number of tasks is increased. To avoid increasing the number of tasks to a number from which we have previously decreased, the algorithm uses the change history (Lines 4-9). If the number at which we want to jump was already used and it wasn't useful, i.e., the algorithm decreased the number of tasks afterwards, a number is chosen such that is at the half of the distance between this number and the current task number. The process is repeated until the maximum reachable value is the current number of tasks incremented by one.

Task number decrease Algorithm 3 describes how the number of tasks is decreased. To cope with a situation in which an aggressive increase in tasks' number leads to contention, the algorithm decreases the number of tasks in the next time period to half of the distance between the current and the previous values (Lines 2-4). The reason of this decision is to jump at a value between the point in which we detect contention and the previous point in which resources were underutilized. If contention is still detected, the algorithm uses the *good_points* history to choose the highest number of tasks for which no contention was detected (Lines 6-7). Note that contention can appear during application runtime due to non-local factors too, e.g., shared access to remote disk, or switch. Thus, a previous high number of tasks for which contention was not detected, might yield contention at the current run.

4 Implementation

Weasel is implemented in Python. For the communication between the workers and the resource manager we use ZeroMQ a distributed message passing library. As an in-memory file system we use MemFS, which depends on redis. MemFS provides a POSIX interface to store files in a distributed simmetrical manner on all the nodes on which the application is running.