

Integrated Deadline Scheduling with MapReduce Tasks in Hadoop

Sunita Ashok Patil¹ Prof. J Geetha²

^{1,2}Department of Computer Science & Engineering

^{1,2}M S Ramaiah Institute of Technology, Bangalore, Karnataka 560054y

Abstract— The MapReduce model implemented by Hadoop is the standard platform for applications processing Big Data. Schedulers are critical in enhancing the performance of MapReduce/Hadoop with limited computing resources in presence of multiple jobs with different characteristics and performance goals. The commonly used schedulers include the First In First Out (FIFO), Capacity, Fair Scheduler etc. But none of the above schedulers consider the strong dependency between the map and reduce tasks. They fail to jointly optimize the placement of map and reduce tasks. They even ignore the data locality for reduce tasks and the deadline constraints for jobs. The Fair Scheduler is not fair to reduce tasks and might cause starvation when allocating excess resources to reduce task without jointly optimizing with map task. The delay scheduling might lead to under utilization of resources at times. Considering all of the above problems, we propose a resource aware Integrated Preemptive Deadline Scheduler with the MapReduce Task Coupling which jointly optimizes the map and reduce tasks by coupling their progress and ensures deadline completion of jobs. The integrated approach ensures fast and under deadline completion of jobs. It mitigates job starvation and makes better slot utilization for reducing the total completion time of jobs. Thus the integrated scheduler improves data locality, slot utilization, mitigates starvation and ensures under deadline completion.

Key words: Hadoop, Mapreduce, Scheduling Algorithm, Data Locality, Deadline Constraints

I. INTRODUCTION

Hadoop is a popular open source implementation of the MapReduce framework that has attracted a great amount of interest from industry and academia [1]. It is an open source platform designed for scalable, reliable, distributed computing. Massive amount of data are computed using a distributed programming model called MapReduce. It is an execution framework for large scale data processing. It consists of 2 main subsystems: HDFS – a distributed file system, and MapReduce – a framework to process large distributed data sets. The Hadoop framework works as follows: A single master manages a number of slaves. An input file, present in the HDFS gets split into even-sized chunks. Then the job is divided by hadoop into a set of tasks and assigns the mapper tasks to a number of workers which resides on the slave nodes for processing. The result of a map task is a set of intermediate key/value pairs. The master is responsible for forwarding the locations of the intermediate key/value pairs to the reduce workers. The reduce workers then contact the slaves, read the required intermediate data, perform the reduce computations and store the results to an output file.

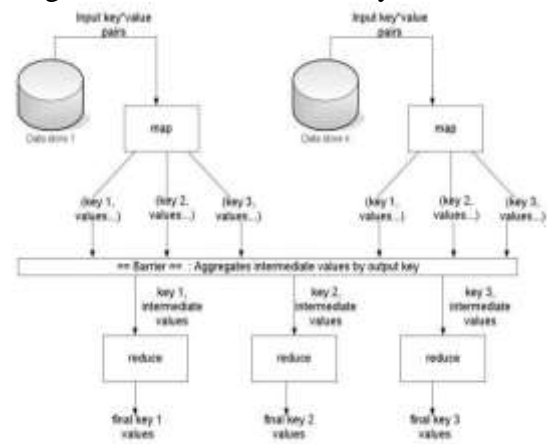


Fig. 1: A MapReduce Computation Diagram

II. RELATED WORK

FIFO is the default Scheduling algorithm where jobs were executed in the order of their submission.

A. FIFO

The default Hadoop scheduler operates using a FIFO queue. A job is partitioned into individual tasks, then they are loaded into the queue and assigned to free slots as they become available on TaskTracker nodes. Although there is support for assignment of priorities to jobs, this is not turned on by default. Typically each job would use the whole cluster, so jobs had to wait for their turn. Even though a shared cluster offers great potential for offering large resources to many users, the problem of sharing resources fairly between users requires a better scheduler. Production jobs will complete in a timely manner, while allowing users who are making smaller ad hoc queries to get results back in a reasonable time [2].

B. Fair Scheduler

The Fair Scheduler aims to give every user a fair share of the cluster capacity over time. Jobs can be assigned to the pools by users and each pool is allocated a guaranteed minimum number of Map and Reduce slots. The Fair Scheduler supports preemption, so if a pool has not received its fair share for a certain period of time, then the scheduler will kill tasks in pools running over capacity in order to give the slots to the pool running under capacity. On certain pools, priority settings can be enforced by administrators. Therefore, tasks are scheduled in an interleaved manner based on their usage of pool, their priority within the pool and capacity of the cluster. As jobs have their tasks allocated to Task Tracker slots for computation, the scheduler tracks the deficit between the amount of time actually used and the ideal fair allocation for that job. As slots become available for scheduling, the next task from the job with the highest time deficit is assigned to the next free slot. Over time, this has the effect of ensuring that jobs receive roughly equal amounts of resources. Shorter jobs are allocated sufficient

resources to finish quickly. At the same time, longer jobs are guaranteed to not be starved of resources[3].

C. Capacity Scheduler

Capacity Scheduler was originally developed at Yahoo to ensure a fair allocation of computation resources amongst users. The scheduler in this case allocates the jobs to the queues with configurable number of Map and Reduce slots. Within a queue, scheduling operates on a modified priority queue basis with specific user limits, with priorities adjusted based on the time a job was submitted, and the priority setting allocated to that user and class of job. When a Task Tracker slot becomes free, the queue with the lowest load is chosen, from which the oldest remaining job is chosen. A task is then scheduled from that job. Overall, this has the effect of enforcing cluster capacity sharing among users, rather than among jobs, as was the case in the Fair Scheduler [2].

D. Delay scheduler

Fair scheduler is developed to allocate fair share of capacity to all the users. By following the fair scheduler two problems are identified i.e Head-of-line scheduling and sticky slots. The first locality problem occurs in small jobs (jobs that have small input files and hence have a small number of data blocks to read). When a job reaches the head of the list which is sorted for scheduling, one of its task is executed on the next slot which is available. The other locality problem, sticky slots, is that there is a tendency for a job to be assigned the same slot repeatedly. The problems aroused because following a strict queuing order forces a job with no local data to be scheduled.

The Head of line problem, can be overcome by scheduler launches a task from a job on a node without local data to maintain fairness, but violates the main objective of MapReduce that schedule tasks near their input data. Running on a node that contains the data (node locality) is most efficient, but when this is not possible, running on a node on the same rack (rack locality) is faster than running off-rack. Delay scheduling is a solution that temporarily relaxes fairness to improve locality by asking jobs to wait for a scheduling opportunity on a node with local data. When a node requests a task, if the head-of-line job cannot launch a local task, it is skipped and looked at subsequent jobs. However, if a job has been skipped long enough, non-local tasks are allowed to launch to avoid starvation. The key insight behind delay scheduling is that although the first slot we consider giving to a job is unlikely to have data for it, tasks finish so quickly that some slot with data for it will free up in the next few seconds [2].

III. PROPOSED WORK

The methodology of our proposed work can be divided into 2 modules:

- Deadline Scheduling
- Checkpoint Scheduling

A preemptive approach for scheduling jobs in an effective way is proposed so that the total completion time of the jobs under deadlines is minimized.

The main idea of our scheduling algorithm is to check for its deadline constraints. The two main factors

taken into consideration for scheduling are completion time and slot utilization.

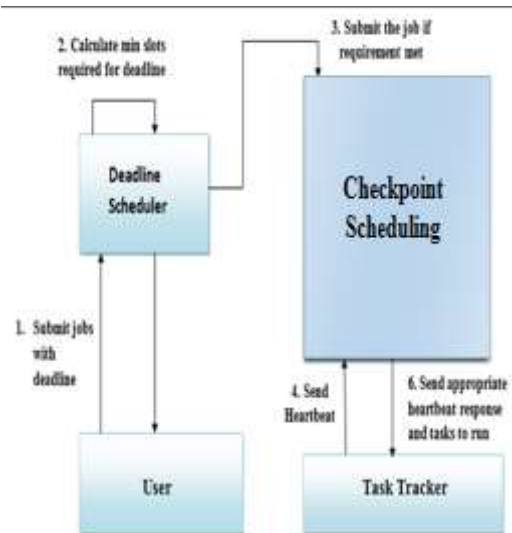


Fig. 2: System Design

A. Completion Time

When a submitted job has required available slots to meet the deadline, the job is scheduled to run. The scheduler first checks for the available slots and then assigns them to the map tasks. It doesn't assign slots to reduce tasks until all map tasks are completed. The completion time of a job is defined as the time when the last reduce task finishes. For example, when a job waits for the completion of the last map task despite other map tasks are finished, the reduce task cannot be executed until the last map task are finished. The main reason is that the slots assigned to all the map tasks are not released until all these tasks are finished. The total completion time in this case has been increased. These kinds of issues are addressed here by changing the policy: if a map task is finished, the slot allocated to the map task is released and the output produced by this map task is sent to a reduce task. By adapting this new policy, the overall completion time is decreased.

B. Slot Utilization

A minimum number of slots to each job is allocated in order to keep more available slots for the subsequent job requests. However, this scheduler has degraded the slot utilization. For example, there are 100 available slots in the Hadoop. A submitted job needs 40 slots. Because the minimum requirement is 20 slots for this job, the scheduler only allocate 20 slots regardless other 20 slots are idle. A preemptive scheduler is used so that it effectively utilizes the slots and avoids the starvation issue.

C. Deadline Scheduling

Definition: $J = (M, R, A, D)$ means a MapReduce job. M , R , A and D denotes Map task set, Reduce task set, job arrived time, and job deadline constraints respectively.

In MapReduce, the job's execution progress includes Map and Reduce stage. So, the job's completion time contains Map execution time and Reduce execution time. In view of the differences between Map and Reduce's code, we divide the scheduling progress into two stages, namely Map stage and Reduce stage. Previous researches usually simplify the Map and Reduce as the same type of

scheduling problem. It may do simplify the problem, but it is improper for the reason that the execution of Map and Reduce's code is different. In the aspect of the task's scheduling time prediction, the execution time of Map and Reduce is not correlative; their execution time depends on the input data and function of their own. Therefore, the scheduling algorithm sets two deadlines: map-deadline and reduce-deadline. And reduce-deadline is just the users' job deadline.

In order to get map-deadline, we need to know the Map task's time proportion on the task's execution time. In a cluster with limited resources, Map slot and Reduce slot number is decided. For an arbitrary submitted job with deadline constraints, the scheduler has to schedule reasonable with the remaining resources in order to assure that all jobs can be finished before the deadline constraints.

Let M_p and R_p be the proportion of Map and reduce task's execution time respectively and $M_p+R_p=1$. E_m and E_r are the map and reduce task's execution time. So,

$$M_p = \frac{E_m}{E_m + E_r} \quad \text{and} \quad R_p = \frac{E_r}{E_m + E_r}$$

Now calculate the map-deadline by M_p :

$$M_d = A + (D - A) * M_p$$

Where A and D denotes job arrived time, and job deadline constraints respectively.

According to map-deadline, we can acquire the current map task's slot number it needs; and with reduce-deadline, we can get the current reduce task's slot number it needs.

We estimate the time needed for the remaining task on the lowest level node. By this way, job and the minimum Map slot requirement of job J can be computed as follows:

$$\delta_j^m = \frac{\sum_{1 \leq p \leq k} SM(J, p) * L_p^m + UM(J) * MM(J, 1)}{|M_d - CurrentTime|}$$

Similarly, the minimum Reduce slot number requirement of job J is:

$$\delta_j^r = \frac{\sum_{1 \leq p \leq k} SR(J, p) * L_p^r + MR(J, 1)}{|D - CurrentTime|}$$

Where $UM(J)$ and $UR(J)$ denotes waiting task set of the job J 's Map and Reduce task type respectively. $MM(J, p)$ denotes the completion time of job J 's Map task which runs on the node in p -th level and $SM(J, p)$ and $SR(J, p)$ is the sum of the remaining Map and Reduce task's execution time of job J which running on the p th level nodes.

The scheduling strategy is based on δ_j^m and δ_j^r . The minimum Map and Reduce slot number required of job J can be denoted as δ_j^m and δ_j^r respectively. The symbol δ_j^m reflects that δ_j^m Map tasks should be scheduled at present in order to meet job J 's map-deadline, as well as to meet the reduce-deadline (job deadline)

The scheduling strategy is based on δ_j^m and δ_j^r . The minimum Map and Reduce slot number required of job J can be denoted as δ_j^m and δ_j^r respectively. The symbol δ_j^m reflects that δ_j^m Map tasks should be scheduled at present in order to meet job J 's map-deadline, as well as to meet the reduce deadline (job deadline) δ_j^r Reduce tasks should be scheduled. In the scheduling process, we take δ_j^m and δ_j^r as the basic criteria of priority allocation [4].

Deadline Algorithm

Collection assignTask(TaskTracker t)

$M = t.freeMapSlots$ /* M is the map slot collection */

$R = t.freeReduceSlots$ /* R is the reduce slot collection */

$Q = \varnothing$ /* job queue */

$T = \varnothing$ /* task collection, to be scheduled */

compute the slot requirement for jobs in queue Q

update jobs' priority in queue Q /* according δ_j^m , δ_j^r and time */

resort the jobs in queue Q /* sort jobs in descending order */

for each job $\in Q$ do

count = 0
if count < job.MapSlotRequirementCount
if job exists waiting map task and $M \neq \varnothing$
Task t = job.obtainMapTask()
 $T = T \cup \{t\}$
remove a Map slot in M
count++
else break

else break
for each job $\in Q$ do
count = 0

if count < job.ReduceSlotRequirementCount
if job exists waiting map task and $R \neq \varnothing$
Task t = job.obtainReduceTask()
 $T = T \cup \{t\}$
remove a Reduce slot in R
count++
else break

else break

return T

Thus the MapReduce scheduling flow in Hadoop is as follows:

- The scheduling maintains a job queue
- The work node, which is called TaskTracker, asks the scheduler for tasks periodicity
- When a request arriving, the scheduler decides which task to be assigned to the TaskTracker according the scheduling algorithm. In this algorithm, we firstly compute δ_j^m and δ_j^r for each job in queue and resort the jobs' order, then we schedule the waiting Map and Reduce tasks respectively. Finally, return the task collection T to the TaskTracker and the TaskTracker will execute the tasks [4].

D. Checkpoint Scheduling

For those jobs that such prediction actually underestimates their execution time, checkpointing avoids wasting the amount of work already performed, yet maintains the high priority of the job holding advance reservation. In checkpoint scheduling, before the execution of one Mapper task, the algorithm will create a local checkpoint file and a global index file. The local checkpoint file is responsible for recording the progress of the current task, and could avoid reexecution from the beginning after task failure. If local task failure happened, the node could restart the task from recent status with the help of local checkpoint file. While the global index is responsible for recording the characteristics of the current execution, thereby help reconstructing the intermediate results in other nodes when the node failed, thus reduces the re-execution time.

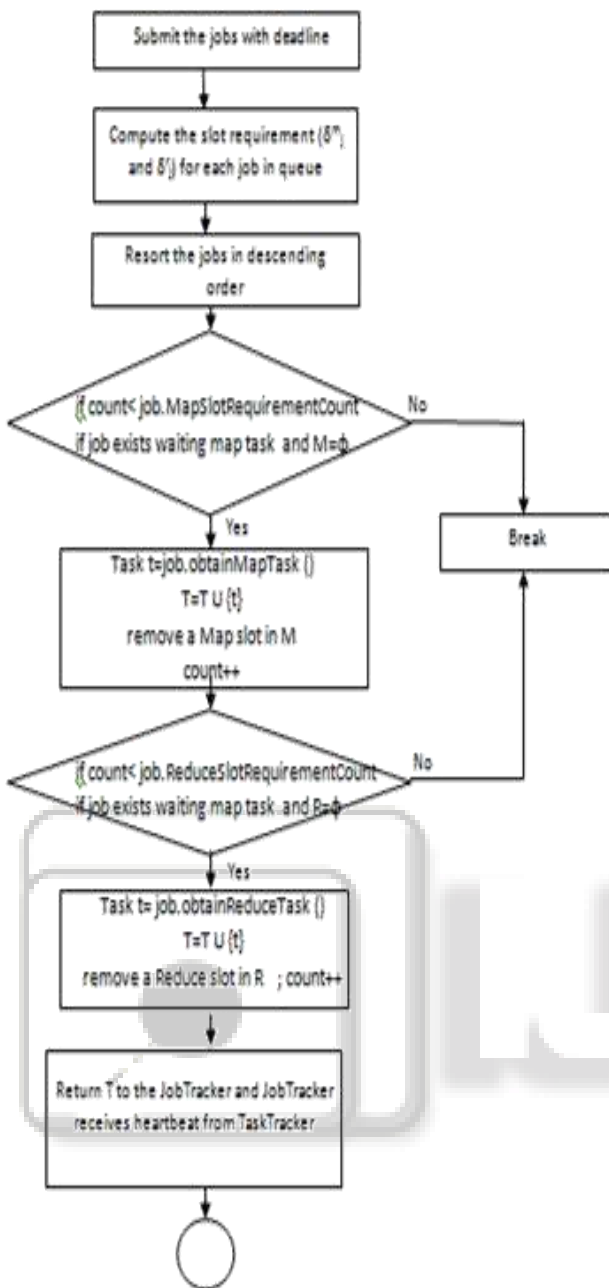


Fig. 3(a): The basic flow of the proposed algorithm

The global index file could be implemented as a checkpoint file saved in HDFS, thus could be accessible in case of node failure. When a task failure occurs, simply read the checkpoint file saved in the local disk, restore task status to the checkpoint, and reload the intermediate results generated before failure, so the duplicated execution can be avoided. When a node failure occurs, the scheduler on the master node is responsible for rescheduling the interrupted mapper tasks to available replica nodes. The replica node can quickly construct the intermediate results of failed tasks according to the global index file, greatly reducing the re execution time. Note if tasks and nodes failed before saving checkpoint, the progress will continue from the recent checkpoint. And if the action of saving checkpoint failed, the progress will start from the recent checkpoint again. The simple failover strategy is effective in distributed computing environment with relatively low cost. If node failure happens in the Reducer stage, then tasks on the failed node are rescheduled to any available replica node. The needed

intermediate results have been copied to the replica node when Mapper tasks finished, thus there is no need to repeat the mapper tasks on the failed node, so the overall completion time of the MapReduce job is greatly reduced. The checkpoint method provides better performance with low overhead when failure happens. Both node and task failure can be supported and failure introduced delay is significantly decreased comparing with traditional rescheduling in Hadoop. Thus could improve overall performance of a MapReduce job [5].

The basic flow of the proposed algorithm is as shown in figure 3(a) and 3(b).

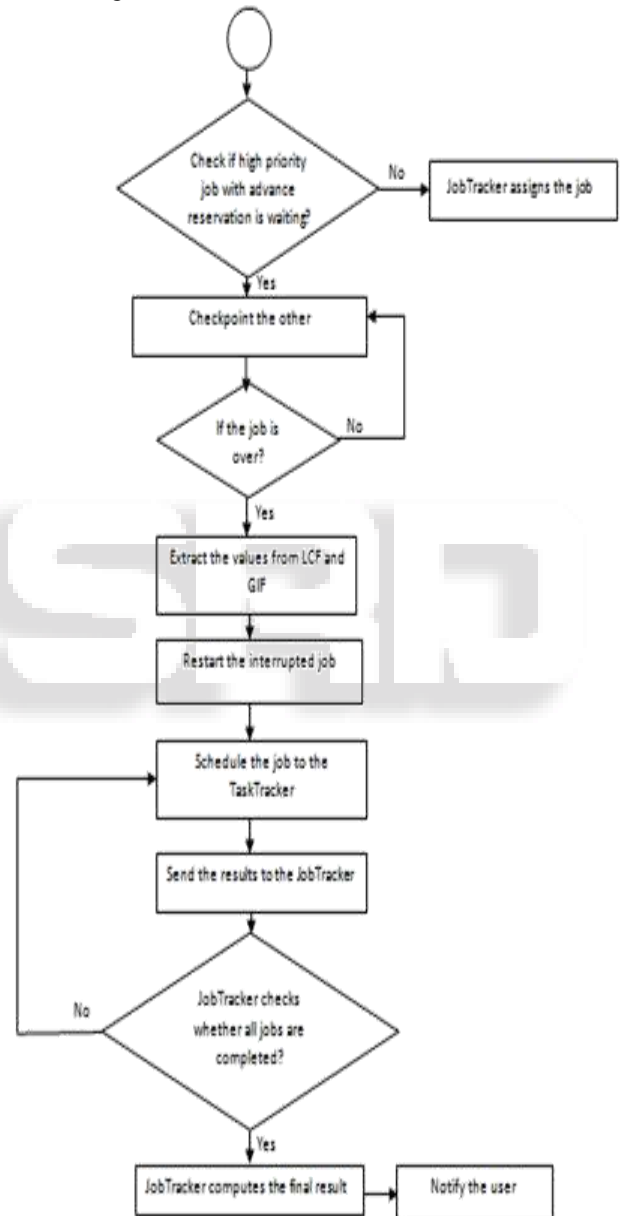


Fig. 3(b): The basic flow of the proposed algorithm

Pseudo code of the algorithm

- 1) User submits the jobs with deadlines.
- 2) The deadline scheduler calculates the minimum slots required i.e. δ_j^m and δ_j^r . If yes, go to Step 3, else notify the user.
- 3) Resort the jobs in descending order.
- 4) Schedule the map and the reduce tasks.
- 5) Return the task collection T to the JobTracker.

- 6) JobTracker receives heartbeat from the TaskTracker and creates 2 files i.e. local checkpoint file and global index file.
- 7) Check if high priority job with advance reservation is waiting.
- 8) If yes, checkpoint the running job, else go to step 12.
- 9) Check if the job (advance reserved) is completed.
- 10) If yes, extract the values from the local checkpoint file and global index file. Else continue to checkpoint the job.
- 11) Once the values are extracted, now restart the interrupted jobs i.e. the pre-empted job is now executed. (This avoids re-execution of the jobs in case of any node failures).
- 12) Schedule the jobs to the TaskTracker.
- 13) Send the results to the JobTracker.
- 14) The JobTracker now checks whether all jobs are completed.
- 15) If yes, the JobTracker computes the final result. Else go to step 12.
- 16) Once the final result is computed, notify the user.

http://hadoop.apache.org/docs/stable/fair_scheduler.html [Dec 10, 2014]

- [8] Capacity Scheduling,
<http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html> [Dec 10, 2014]
- [9] Hadoop's Capacity Scheduler,
http://hadoop.apache.org/core/docs/current/capacity_scheduler.html. May. 13, 2006 [Nov. 15, 2014].
- [10] Jian Tan, Xiaoqiao Meng, Li Zhang, "Coupling Task Progress for MapReduce Resource-Aware Scheduling" Proc. of IEEE, Vol.1, 2013

IV. CONCLUSION

Thus a resource aware Integrated Preemptive Deadline Scheduler with the MapReduce Tasks which jointly optimizes the map and reduce tasks by coupling their progress and ensures deadline completion of jobs is proposed. It is an integrated approach that ensures fast and under deadline completion of jobs, by the sequential cooperation between map and reduce tasks. It mitigates job starvation and makes better slot utilization for reducing the total completion time of jobs. It helps in maximizing job completion under deadline.

REFERENCES

- [1] Shen Li, Shaohan Hu, Shiguang Wang, LuSu, Tarek Abdelzaher, Indranil Gupta, Richard Pace "WOHA: Deadline-Aware Map-Reduce Workflow Scheduling Framework over Hadoop Clusters."
- [2] B.Thirumala Rao, Dr. L.S.S.Reddy "Survey on Improved Scheduling in Hadoop MapReduce in Cloud Environments", International Journal of Computer Applications (0975-8887) Volume 34-No.9, November 2011
- [3] Mr.A.U.Patil, Mr.T.I Bagban, Mr.A.P.Pande "Recent Job Scheduling Algorithms in Hadoop Cluster Environments: A Survey" International Journal of Advanced Research in Computer and Communication Engineering, Vol. 4, Issue 2, February 2015
- [4] Zhuo Tang, Junqing Zhou, Kenli Li, Ruixuan Li, "A MapReduce Task Scheduling algorithm for deadline constraints" Cluster Computing, Springer, January 2013
- [5] Yang Liu, Wei Wei, Yuhong Zhang "Checkpoint and Replication Oriented Fault Tolerant Mechanism for Map Reduce Framework", TELKOMNIKA Indonesian Journal of Electrical Engineering Vol.12, No.2, February 2014, pp. 1029 ~ 1036
- [6] Hadoop's Fair Scheduler,
http://hadoop.apache.org/common/docs/r0.20.2/fair_scheduler.html Feb. 25, 2008 [Nov. 3, 2014].
- [7] Fair Scheduling,