`LowCost`: A Cost-Based Reservation Algorithm for HADOOP YARN++

Carlo Curino, Subru Krishnan, Ishai Menache, Seffi Naor, Sriram Rao, and Jonathan Yaniv

[1] CISL, Microsoft, ccurino@microsoft.com
[2] CISL, Microsoft, subru@microsoft.com
[3] Microsoft Research, ishai@microsoft.com
[4] CS Department, Technion, naor@cs.technion.ac.il
[5] CISL, Microsoft, sriramra@microsoft.com
[6] CS Department, Technion, jyaniv@cs.technion.ac.il

1 Summary

Using the Rayon [1] abstraction, we design, implement and evaluate `LowCost` – a new cost-based planning algorithm for HADOOP YARN++. Through simulations and experiments on a test cluster, we show that `LowCost` substantially outperforms the existing algorithm (`Greedy`) on a variety of important performance metrics.

2 Background - The Scheduling Model

Reservation requests are encoded using the Reservation Definition Language (RDL). RDL expressions are composed from basic requests, called *atoms*, and a set of allowed operators.

RDL Atom. An RDL atom a represents the resource requirements of a single atomic task (stage). Each atomic task requires a number of allocation units, called *containers*. The RDL atom a specifies several requirements on the requested containers, which are encoded through the following parameters.

- *capability:* the required capability from a single container, given as a multi-dimensional bundle of resources, e.g., $\langle$8GB RAM, 4 cores$\rangle$.
- *duration:* minimum lease duration of a single container; each container must persist for at least *duration* time steps.
- *numContainers:* the total amount of containers requested by a.
- *gang (concurrency):* parallelism requirements on the allocation; the number of containers allocated to a at each time must be an integer multiplication of *gang*.

RDL Expressions. The planner accepts the following type of RDL requests:

- $\mathsf{any}(a_1, \ldots, a_n)$: requires that at least one of the atomic requests $a_1, \ldots, a_n$ is satisfied.
- $\mathsf{all}(a_1, \ldots, a_n)$: requires that all the atomic requests a_i are satisfied.
- $\mathsf{order}(a_1, \ldots, a_n)$: A dependency expression which requires for every i that the allocation of a_i strictly precedes the allocation of a_{i+1}.

3 Planning Algorithms

We first describe a general planning mechanism which is common to all algorithms; see Algorithm 1 for pseudo-code. Each job request submitted to the general planner is composed of an RDL expression e and a time interval I during which the job should be allocated. The general planner treats job requests differently, depending on the type of e. An order request is placed by allocating the atomic tasks in reverse topological order. Each atom is placed using the **PlanAtom** procedure, which receives an RDL atom r and a subinterval I_r during which r is allocated. The planner enforces dependencies between atoms by setting the right endpoint of I_r as the starting time of the successor of r. The specific implementation of **PlanAtom** is algorithm-specific, and hence described separately for each algorithm in the subsequent sections. The general planner handles all requests in a similar manner, except that I_r is set to I for each atom. For any requests, the planner simply selects the first alternative which is successfully allocated by **PlanAtom**.

Algorithm 1: General Planning Algorithm

```
PlanRDL(Plan p, RDL e, Interval I)
    switch (e.type) do
        case (order)
            t ← I.end()
            for (RDLAtom r : reverse(e.requests) do
                alloc ← PlanAtom(p,e,r,new Interval(I.start(),t))
                if (!alloc) then fail()
                t ← alloc.start()
        case (all)
            for (RDLAtom r : reverse(e.requests) do
                alloc ← PlanAtom(p,e,r,I)
                if (!alloc) then fail()
        case (any)
            for (RDLAtom r : reverse(e.requests) do
                alloc ← PlanAtom(p,e,r,I)
                if (alloc) then success()
            fail();
    success();
```

3.1 Greedy

Currently, Rayon implements the following `Greedy` allocation rule (Algorithm 2). Given an atomic request r and an interval I, the algorithm allocates gangs of r starting from the rightmost possible duration interval in I, while meeting the capacity constraints of the plan.

`Greedy` is optimal for a single job request. Namely, for every job represented by an RDL expression e: if there exists a feasible allocation of e, then `Greedy` is guaranteed to succeed in allocating e. However, using `Greedy` for multiple requests has several shortcomings:

Algorithm 2: `Greedy`

PlanAtom.Greedy(Plan p, RDL e, Atom r, Interval I)
  $t \leftarrow I$.end
  while (r is not fully allocated) **do**
    if (t $< I$.start) **then** fail()
    allocate as many gangs in the interval $[t - r.\text{duration}, t]$
      without exceeding the capacity of p
    $t \leftarrow t - 1$
  return p.allocation(r)

1. `Greedy` is not aware of the plan "state". This can cause the algorithm to allocate jobs during loaded intervals, instead of spreading the plan allocation more evenly throughout time.
2. `Greedy` generates tall and skinny allocations of atomic tasks. This significantly increases the amount of task preemptions, and moreover, causes the allocation to become more sensitive to outliers.
3. `Greedy` creates "peaks" in the global plan allocation, which might prevent future requests from being allocated.

3.2 Cost-Based Planning Algorithms

Our algorithms rely on a cost-based approach: each time slot t is associated with a cost $c(t)$. The cost $c(I)$ for a time interval I is defined as the total cost of time slots within the interval. The cost function $c : \mathbb{N} \to \mathbb{R}$ typically represents the current state of the cluster. Intuitively, high cost indicates a busy time period. However, more generally, cost functions can also incorporate expected future demand, projected capacity changes, etc. In our current implementation, the cost function represents the dominant resource (DR),

$$c_{\text{DR}}(t) = \max \left\{ \frac{p.\text{load}(t).\text{mem}}{p.\text{capacity}(t).\text{mem}}, \frac{p.\text{load}(t).\text{cores}}{p.\text{capacity}(t).\text{cores}} \right\}. \quad (1)$$

We note that finding an allocation for a single RDL expression e that minimizes the total incurred cost is NP-hard in general, even when the cluster consists of a single resource. Nevertheless, the algorithms we develop produce a "good enough" global allocation without requiring to find the optimal cost-effective allocation per request.

We now present `LowCost`, our cost-based planning algorithms. We note that `LowCost` has different variants, which we describe below. Similar to `Greedy`, `LowCost` follows the general planning scheme described in Algorithm 1. The difference between the algorithms is in the **PlanAtom** phase (Algorithm 3): The `LowCost` algorithm for allocating a single RDL atom r consists of two phases. The purpose of the first phase is to identify a subinterval I_r of I in which r will be allocated. This phase is only required for order requests, since its purpose is to verify that the chain dependencies are preserved; for any other request, we simply set $I_r = I$. Recall that the atoms of an order request are allocated in reverse order, and that when **PlanAtom** is called with an atom r and interval I, all succeeding atoms have already been allocated later than I.

The first phase essentially partitions I between all atoms preceding r (including r), such that each atom receives a weighed fraction of I; the weight of an atom corresponds to its total requested workload. Each atom is then allocated within its segment. Specifically, I_r is set as

Algorithm 3: `LowCost`

PlanAtom.LowCost(Plan p**,** RDL e**,** Atom r**,** Interval I**)**
- **Phase 1:**
  - $t \leftarrow$ **GetAtomEarliestStartTime**(p,e,r,I)
  - $I_r \leftarrow$ **new** Interval(t, I.end)
- **Phase 2:**
  - **return AllocateAtom**(p,e,r,I_r)

GetAtomEarliestStartTime(Plan p**,** RDL e**,** Atom r**,** Interval I**)**
- **if** (e.type != order) **then return** I.start
- $R \leftarrow$ all atoms in e.atoms up to r (including)
- $tot_{\text{noDur}} \leftarrow I.\text{end} - I.\text{start} - \sum_{r' \in R} r'.\text{duration}$
- $\ell \leftarrow r.\text{duration} + tot_{\text{noDur}} \cdot \frac{\text{weight}(r)}{\sum_{r' \in R} \text{weight}(r')}$
- **return** (I.end - ℓ)

AllocateAtom(Plan p**,** RDL e**,** Atom r**,** Interval I_r**)**
- **while** (r is not fully allocated) **do**
  - $S \leftarrow$ collection of intervals of length r.duration contained in I_r
  - $DI \leftarrow \text{argmin}_{DI \in S}\{\, c_{\text{DR}}(DI) \mid \text{a gang allocated in } DI \text{ can fit in } p \,\}$
  - **if** ($!DI$) **then** fail()
  - p.allocate(r, r.gang * r.capability, DI)
- **return** p.allocation(r)

the segment corresponding to r when **PlanAtom** allocates r. In case an atom does not utilize all of its segment, the partition is updated dynamically.

The second phase of the algorithm allocates a single atom within the segment I_r assigned to it. This is done iteratively. In each iteration, the algorithm considers a set S of allocation options for a single gang, i.e., an interval of length r.duration contained in I_r. We refer to each such option as a *duration interval*. Variants of the `LowCost` algorithm set S as follows.

- `LowCost-E` (E-Exhaustive): The set S contains all of the durations intervals in I that can fit a gang.
- `LowCost-S` (S-Sample): The set S is a sample of N duration intervals contained in I that can fit a gang.
- `LowCost-A` (A-Aligned): The set S contains duration intervals that can fit a gang and are of the form $[I.\text{end} - i \cdot r.\text{duration}, I.\text{end} - (i+1) \cdot r.\text{duration}]$ for any i that defines an interval within I.

If S is empty, then the algorithm fails. Otherwise, the algorithm finds a duration interval in S of minimal cost, allocates a single gang within that duration interval and repeats until all gangs have been allocated[7] We evaluate the different variants of `LowCost` in Section 4. For now, we provide a qualitative discussion. `LowCost-E` considers all of the possible

[7] In our implementation, in order to reduce running-time, we actually allocate $G > 1$ gangs in every iteration. G is determined as a function of I_r, the total number of gangs that need to be allocated, and the duration of a gang.

duration intervals. This minimizes the cost of allocating the current gang, however it might generate "spiky" allocations due to misalignment between allocated gangs. Moreover, the runtime of this exhaustive approach is significantly larger compared to others. `LowCost-S` could reduce the runtime by sampling only a subset of duration intervals. However, the allocations it produces are imbalanced and discontinuous; further, if the sample size is too small, performance would not be good enough. `LowCost-A` restricts the duration intervals it considers such that all options are "aligned" to each other (i.e., their respective time intervals are disjoint). This prevents the creation of short spikes, results smoother allocations, and is computationally efficient. Intuitively, although not all options are examined, `LowCost-A` attempts to provide adequate coverage of the available options in I.

Finally, we note that unlike `Greedy`, the cost-based algorithms described here do not guarantee that an RDL request can always be satisfied, given that a feasible allocation for it exists. To recover this property, all variants of `LowCost` run `Greedy` if they initially fail to allocate an RDL request.

4 Evaluation

In this section we evaluate our algorithms against `Greedy`. This section is organized as follows. We first outline our evaluation methodology (Section 4.1). We then describe the main results of our simulations (Section 4.2) and the experiments on a test cluster (Section 4.3).

4.1 Methodology

The basic setting of our evaluation involves jobs that arrive online, each with a reservation requirement specified via RDL. The algorithm handles one job at a time. It either finds a feasible allocation for the job (which satisfies the RDL requirement), or rejects the job. In case it accepts the job, it assigns a reservation to it. The assigned reservation is fixed, i.e., it cannot be transformed later on to another feasible reservation (modifying existing reservations is a future research direction).

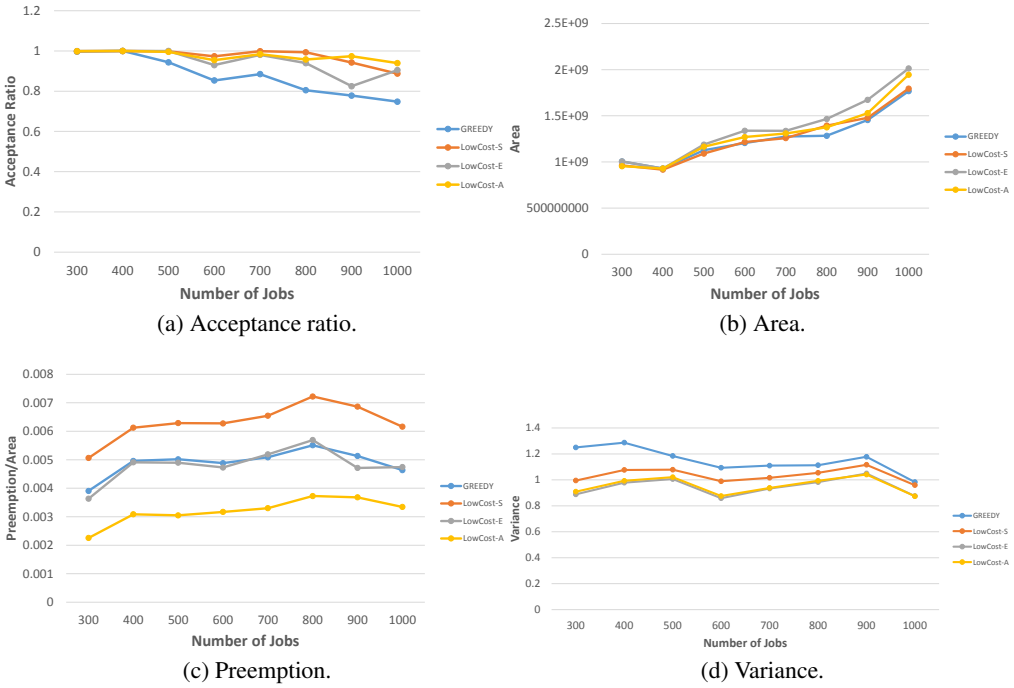
To compare the algorithms, we consider the following metrics.

- *Acceptance Ratio (AR).* The percentage of jobs that have been accepted.
- *Area.* Measures the total containers hours that have been allocated.
- *Preemption.* Measures the variability of the per-job allocation (summed over all jobs). Specifically, we count the decreases in container allocation for the job across time. We sum over all jobs and normalize by Area.
- *Variance.* Measures the variability of the entire plan.
- *Running Time.* The average time it takes the algorithm to handle a job.

4.2 Simulations

The simulations allow us to efficiently test different flavors of `LowCost`. In addition, we are able to control the cluster size, and examine the algorithms in the scale that we aim for in production systems (thousands of machines).

The general setting of each simulation is as follows. We set the total number of jobs and the cluster size. All jobs are generated at time zero (We use Gridmix as the workload). However, they are revealed to the algorithm one by one (i.e., in an online manner), and the



(a) Acceptance ratio.

(b) Area.

(c) Preemption.

(d) Variance.

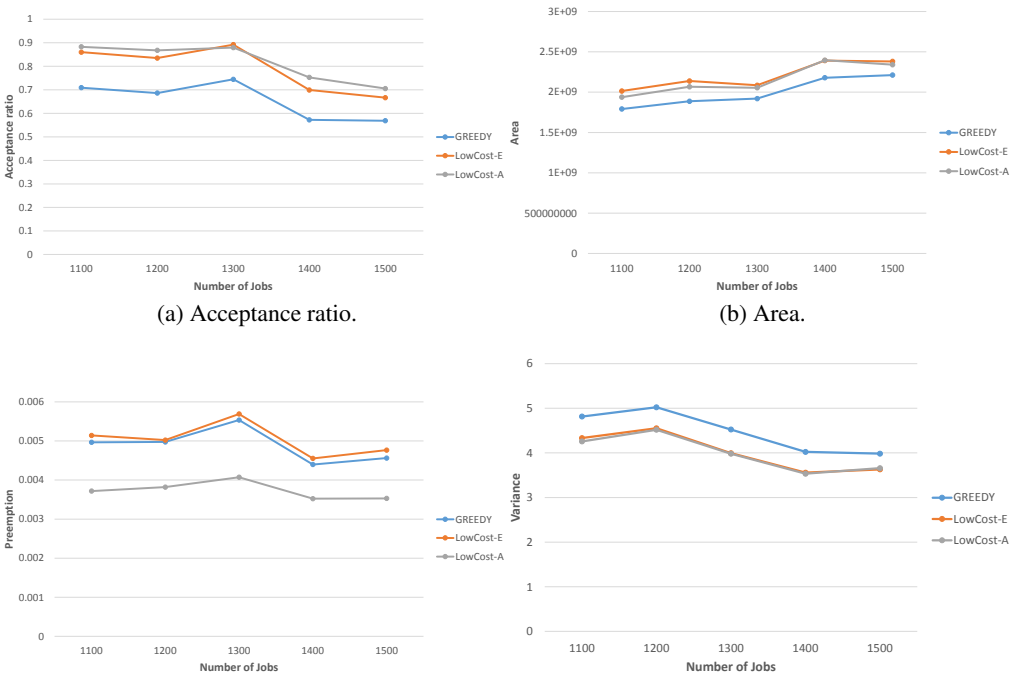
Fig. 1: Simulation results on a 4K cluster as a function of the number of submitted jobs (300-1000). Results averaged over 5 runs.

algorithm has to decide whether to accept the job or reject it. As mentioned above, accepting a job comes with its reservation, which cannot be later changed.

In our first set of simulations we consider a moderate number of jobs (300-1000). The cluster size is set to 4000 machines. Simulation results are summarized in Figure 1. As can be seen from the figure, all variants of `LowCost` perform better than `Greedy` in terms of Acceptance Ratio, Area and Variance. In terms of Preemption, `LowCost-A` is the best, `Greedy` is comparable with `LowCost-E`, and `LowCost-S` is the worse; this is expected since `LowCost-S` may inherently choose slots without time-continuity, which increases the preemption. In terms of runtime (not shown in the figure), `Greedy` is the fastest, while `LowCost-A`, `LowCost-E` and `LowCost-S` are around 4x, 20x and 200x slower, respectively. Because `LowCost-S` turned out too slow, and has not resulted in better performance, we would not consider it further. `LowCost-E` is a bit better than `LowCost-A` in terms of Area, but much slower and also worse in terms of Preemption. Overall, `LowCost-A` seems the most promising so far.

To reinforce our conclusions, we carried out a second set of experiments with a larger number of jobs. Results are summarized in Figure 2. Results remain qualitatively similar, and the improvements compared to `Greedy` are even higher.

Bottom-line: `LowCost-A` is the best algorithm. Compared to `Greedy`, it improves Area by 15%, accepts 30% more jobs, with 20-40% less preemption and 20% less variance. The runtime increase is by less than 5x.



(a) Acceptance ratio.

(b) Area.

(c) Preemption.

(d) Variance.

Fig. 2: Simulation results on a 4K cluster as a function of the number of submitted jobs (1100-1500). Results averaged over 5 runs.

4.3 Real-cluster experiments

We now describe our experiments on a real cluster. We divide the cluster capacity (around 240 nodes) into two equal parts. One part runs `Greedy`, and the other part runs our winning variant of `LowCost`, `LowCost-A`. Throughput this section we refer to the latter algorithm simply as `LowCost`.

The workload we use is based on Gridmix. In order to perform a substantial number of different experiments, we use a modified version of Gridmix in the bulk of our experiments. In particular, we truncate the original time-distributions of Gridmix (which originally captures one week of activity) by nullifying the probability of jobs arriving at a time later than our desired experiment time. We perform two sets of experiments: (i) shorter-term: Jobs arrive within one hour[8]. (ii) longer term experiments: Jobs arrive within $100,000$ seconds (28 hours).

A note on the experiment settings and their limitations. Besides the obvious differences from simulation conditions – such as machines can fail, potential communication issues, etc. – there are yet other notable differences. Rather than all jobs being submitted at time zero, jobs are submitted gradually, as a function of their potential start time. This changes the conditions under which the algorithms operate. Intuitively, there is a bit less flexibility to the algorithm. In addition, the number of machines is relatively small. This implies that a large

[8] Before truncating entries with times greater than one our, we divide the distribution times by 100, to make things more "interesting" closer to time zero.

job ("elephant") can occupy the entire cluster for some time, in which case all algorithms will fail to admit new jobs. Due to the above, one might expect that the gains that we obtain would be less than in the simulation settings in which we had thousands of machines. Nevertheless, we obtain similar gains (and even better ones) in most of the scenarios that we consider below.

Shorter-term experiments We consider both high-load (around 750 submitted jobs) and medium-load (around 140 submitted jobs). The acceptance rate statistics are summarized in Table 1. `LowCost` accepts around %30 (%55) more jobs at medium (high) load. In addition (not seen in the table), we note that `LowCost` strictly dominates `Greedy`, in the sense that it admits more jobs in each individual run.

We point out that the relatively small cluster size prevents us to experiment with low-load scenario in which most jobs are accepted - for that to happen in the one-hour time window, we need to submit a very small number of jobs, which will make the results statistically insignificant.

Algorithm	Medium Load	High Load
`Greedy`	$32.8\% \pm 4.9\%$	$15.7\% \pm 3.1\%$
`LowCost`	$43.3\% \pm 3.1\%$	$24.4\% \pm 5.3\%$

Table 1: Acceptance ratios for `Greedy` and `LowCost`. Results averaged over 5 runs.

Extracting the other performance metrics is tricky. This is because the system keeps track of the metrics only for active reservations. To still get meaningful statistics, we output their values after every ten additional jobs that have been accepted. We summarize the average gains of `LowCost` relative to `Greedy` in Table . Due to the above mentioned limitation, we provide results only for Preemption and Variance, whose instantaneous values are more meaningful than for the Area metric. The results show substantial gains for `LowCost`.

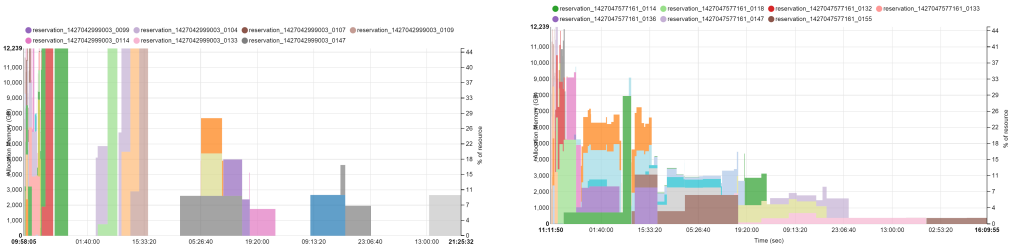
Metric	Medium Load	High Load
Preemption	$46.9\% \pm 9.5\%$	$45.1\% \pm 6.7\%$
Variance	$22.1\% \pm 16.3\%$	$26.9\% \pm 16.7\%$

Table 2: Reductions in Preemption and Variance relative to `Greedy`. Results averaged over 5 runs.

Finally, we zoom-in on an individual run, and show the visualization of the different plans (Figure 3). In general, `Greedy` results in tall and skinny reservations, while `LowCost` leads to shorten, more spread-out reservations. Intuitively, this allows `LowCost` more flexibility to accept new jobs, as less period in times are completely booked.

Longer-term experiments Recall that in the longer-term experiments jobs are submitted over a period of roughly 28 hours. We test medium load conditions by submitting around 950 jobs over that period. The acceptance rate results are summarized in Table 3. As before, `LowCost` obtains better Acceptance Ratio than `Greedy` (although the margins are less than in the shorter-term experiments).

The results for the other metrics are summarized in Table 4, and are fairly similar to what observed before.



(a) Greedy. (b) Lowcost.

Fig. 3: Visualization of the plans at some point throughout the execution of the two algorithms. These snapshots are taken at high-load conditions.

Algorithm	Medium Load
`Greedy`	$48.8\% \pm 0.9\%$
`LowCost`	$54.8\% \pm 1.1\%$

Table 3: Acceptance ratios of `Greedy` and `LowCost` for the longer-term experiments. Results averaged over 3 runs.

Metric	Medium Load
Preemption	$40.2\% \pm 22.4\%$
Variance	$24.6\% \pm 14.8\%$

Table 4: Reductions in Preemption and Variance relative to `Greedy`. Results averaged over 3 runs.

Summary The results obtained in the real-cluster experiment are qualitatively similar to those we got in simulations. We expect similar gains on larger clusters.

References

1. C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

A Class Hierarchy

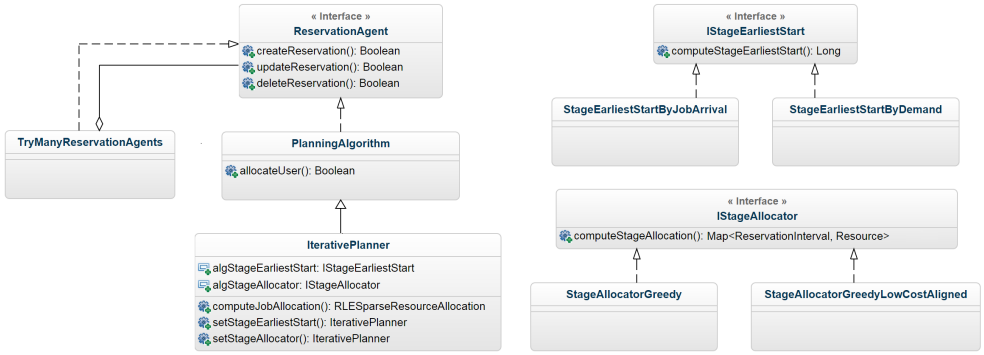


Fig. 4: Class Hierarchy

Class descriptions:

- **ReservationAgent:** The planner interface exposed by Rayon [1].
- **PlanningAlgorithm:** An abstract planning scheme, implements ReservationAgent.
- **IterativePlanner:** A subclass of PlanningAlgorithm, allocates stages in reverse order (this is the general planner; see Algorithm 1).
- **IStageEarliestStart:** An interface for implementations of StageEarliestStart.
- **StageEarliestStartByJobArrival:** Sets the earliest start time of a stage as the job arrival time; used by Greedy.
- **StageEarliestStartByDemand:** Sets the earliest start time of a stage proportional to its resource demand; used by `LowCost`.
- **IStageAllocator:** An interface for implementations of StageAllocator.
- **StageAllocatorGreedy:** An implementation of the `Greedy` stage allocator.
- **StageAllocatorLowCostAligned:** An implementation of the `LowCost` stage allocator.
- **TryManyReservationAgents:** A meta-planner. Implements ReservationAgent by executing a list of planners until one of them succeeds.

The class **AlignedPlannerWithGreedy** implements our "winning " algorithm (evaluated in Section 4). This class holds an instance of TryManyReservationAgents, which first attempts to allocate via a LowCost scheme (IterativePlanner with StageEarliestStartByDemand and StageAllocatorLowCostAligned). If that scheme fails, then the algorithm runs `Greedy` (IterativePlanner with StageEarliestStartByJobArrival and StageAllocatorGreedy).