JellyFish: Online Performance Tuning with Adaptive Configuration and Elastic Container in Hadoop Yarn

Xiaoan Ding, Yi Liu, Depei Qian Sino-German Joint Software Institute
Beihang University
Beijing, China

Main contributions:

-- propose a novel *elastic container* that can expand and shrink dynamically according to resource usage of the container

-- In searching desirable configuration, it uses a divide and conquer approach to reducing the dimensionality of searching space

-- JellyFish firstly tunes configuration parameters to improve task performance, and secondly reschedules idle resources to improve job performance and overall resource utilization in the cluster

Architecture Overview

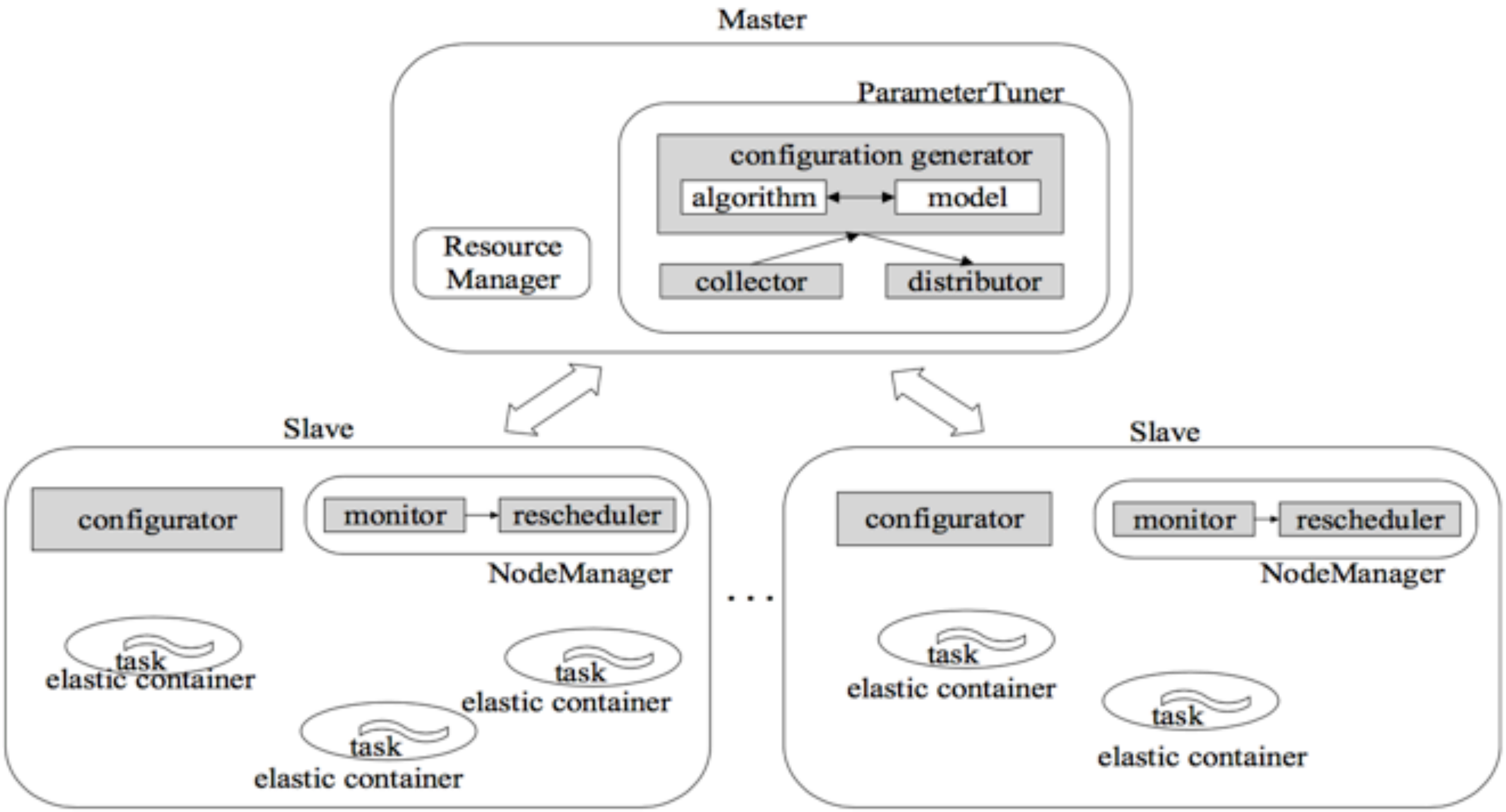


Fig. 1. The architecture of JellyFish.

Two perspective:

---tuning configuration parameters

---rescheduling resources

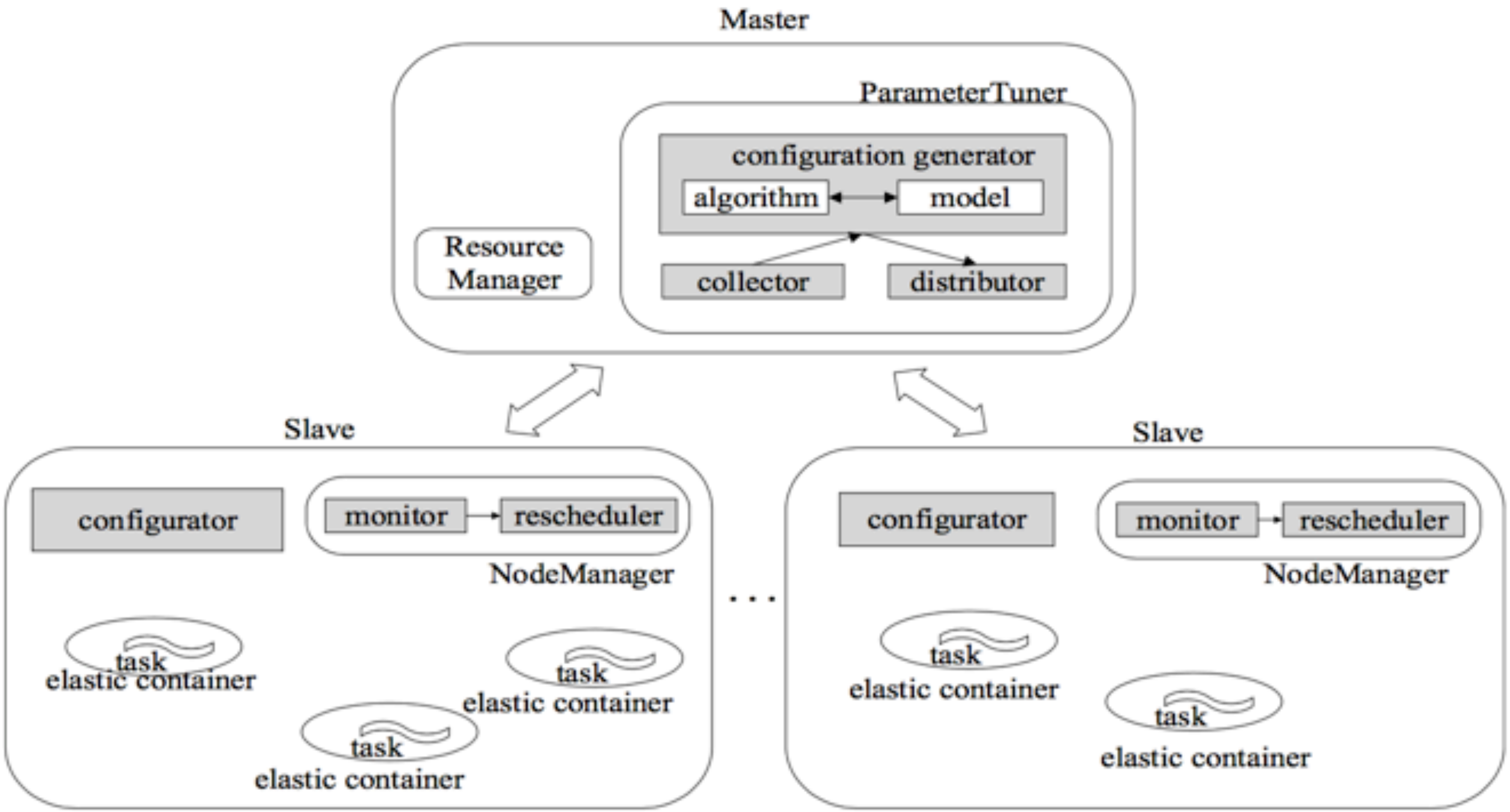


Fig. 1. The architecture of JellyFish.

configuration parameter tuning
--The job-level configuration is changed from constant to dynamic on-the-fly task configuration

--JellyFish collects task statistics, generates and distributes configurations to newly started tasks with ParameterTuner

-- The *generator* collects statistics of running tasks from the *collector*, searches suitable configuration values according to real-time statistics, and passes configurations to the *distributor*

-- The distributor maintains the best configuration at present and a list of test configurations

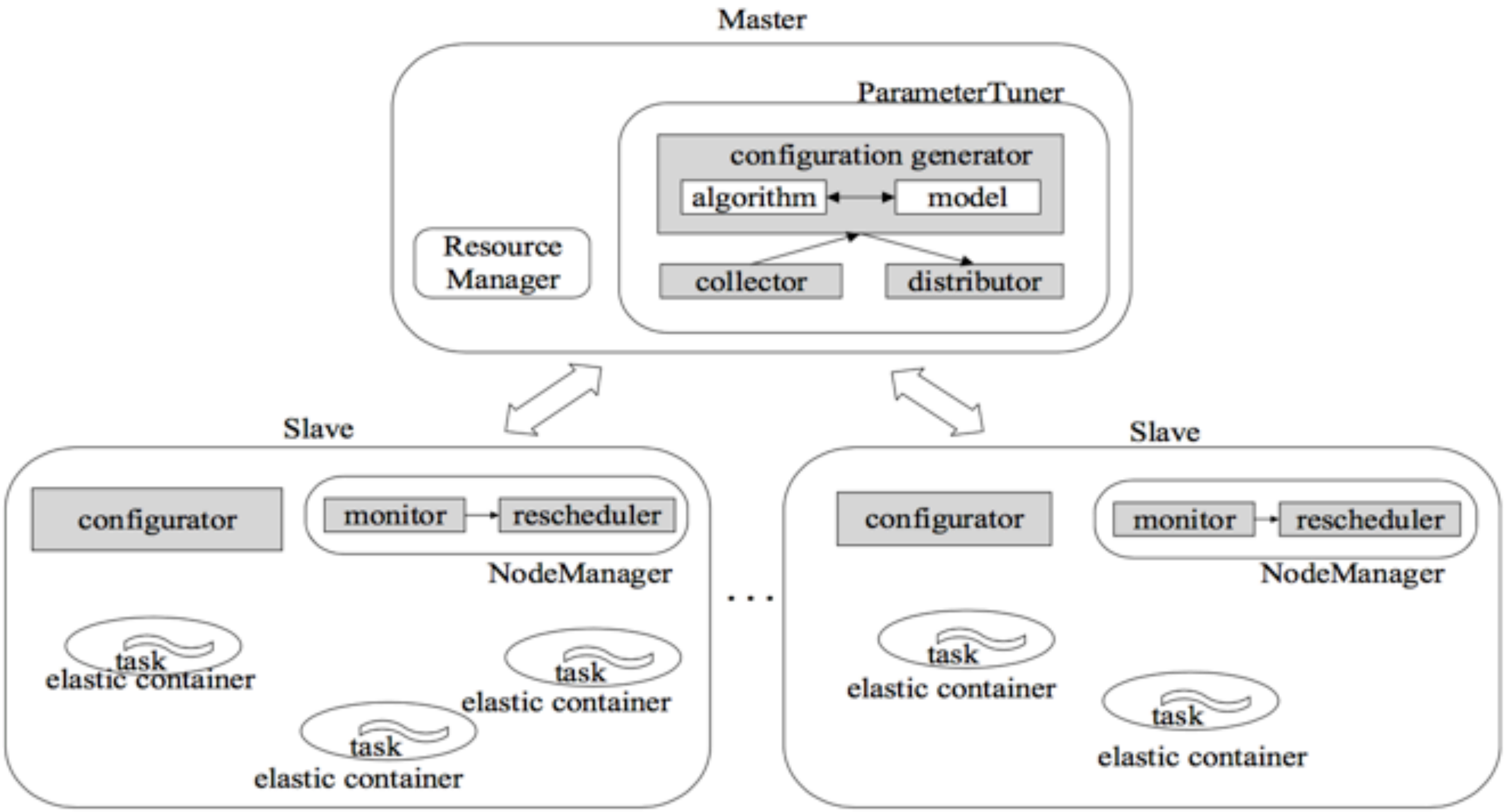


Fig. 1. The architecture of JellyFish.

resource rescheduling
--The system uses novel *elastic container* and a resource re-scheduling strategy to employ idle resources on the node

--each slave nodes has a *monitor* and a *resource rescheduler* that work together with *elastic containers*

--The *monitor* traces all running containers continuously and delivers resource usage of individual container to *resource rescheduler* in real time

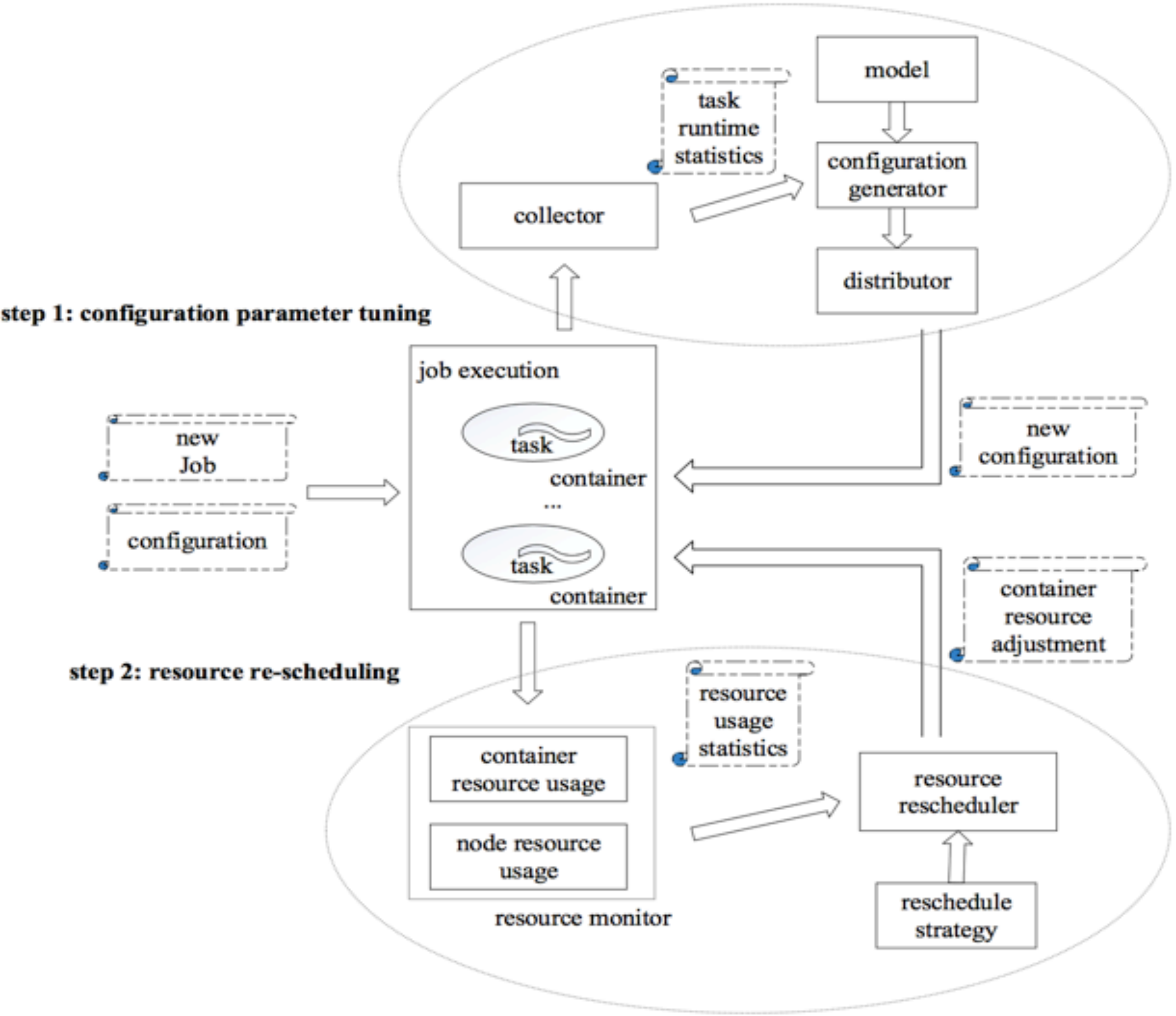


Fig. 2. The tuning process of JellyFish.

--each task runs with a specific configuration.

--The *collector* collects task runtime statistics and sends it to *configuration generator*.

-- Newly generated configurations are placed in *distributor* and assign to new tasks in the next wave.

-- The tuning process iterates until it satisfies the constraints in the algorithm

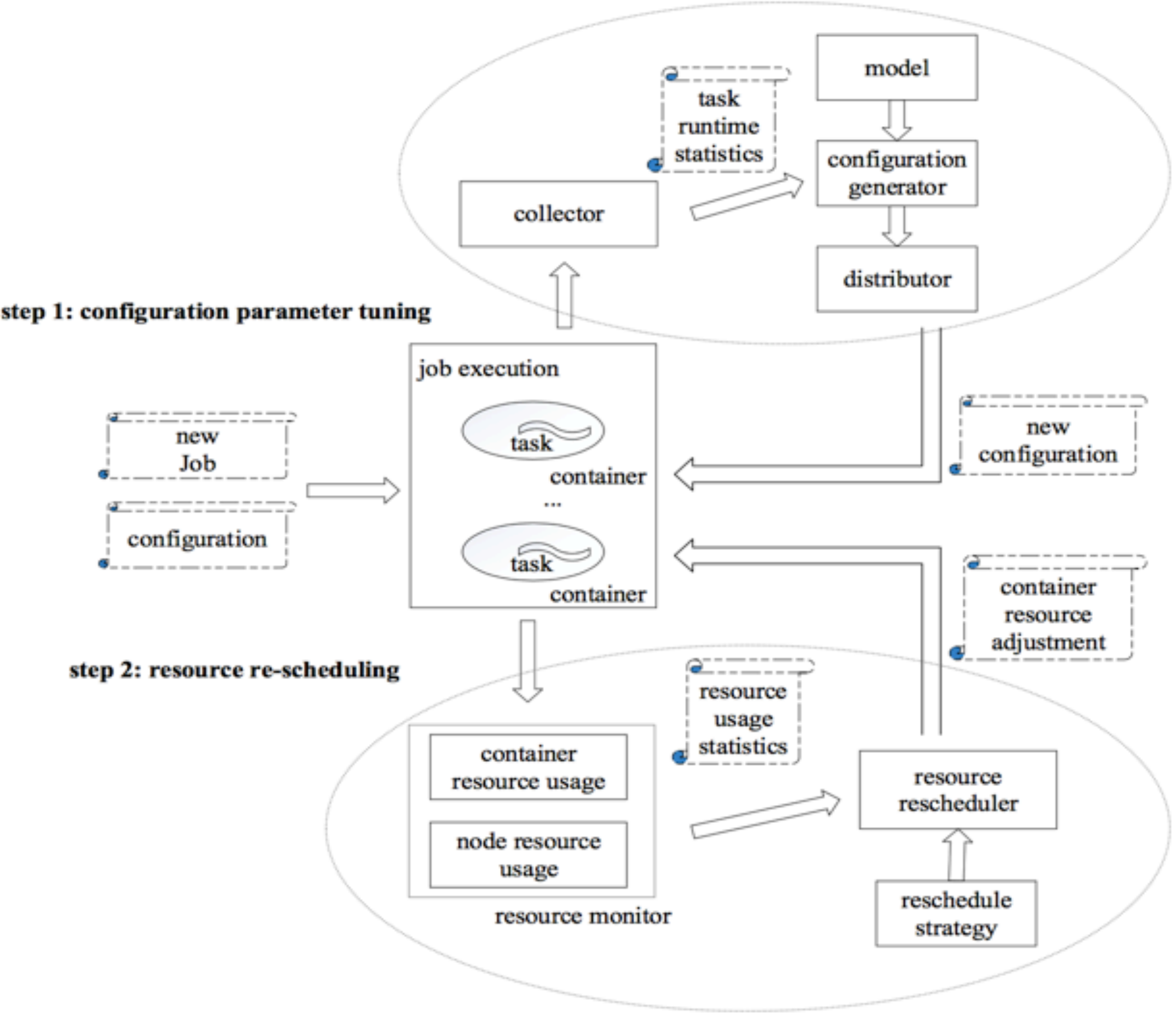


Fig. 2. The tuning process of JellyFish.

--Resource rescheduling process starts only after a fixed configuration has been generated

--the *monitor* tracks runtime resource usage of each container and resource usage in the node

--These statistics are used to decide whether to expand or shrink the capacity of each container based on current strategy

Selecting Tuning Parameters

TABLE I. CONFIGURATION PARAMETER IN JELLYFISH

parameter	default value	symbol
map phase tuning parameter		
mapreduce.task.io.sort.mb	100	p_{mapBuf}
mapreduce.map.sort.spill.percent	0.80	$p_{mapBufSpill}$
mapreduce.map.java.opts	200	$p_{mapJavaOpts}$
reduce phase tuning parameter		
mapreduce.reduce.input.buffer.percent	0.0	$p_{rudInPerc}$
mapreduce.reduce.shuffle.input.buffer. percent	0.70	$p_{shufInBufPerc}$
mapreduce.reduce.shuffle.merge.percent	0.66	$p_{shufMergePerc}$
mapreduce.reduce.shuffle.memory.limit. percent	0.25	$p_{perShufPerc}$
mapreduce.reduce.merge.inmem.threshold	1000	$p_{inmemMergeThre}$
mapreduce.reduce.shuffle.parallelcopies	5	$p_{paraCopy}$
mapreduce.reduce.java.opts	200	$p_{redJavaOpts}$
both map and reduce phase tuning parameter		
mapreduce.task.io.sort.factor	10	$p_{sortFactor}$

-- follow the suggestion from previous studies and only selects the most important ones

Configuration Parameter Tuning

Algorithm 1 Model-based hill Climbing

```
Initialize sampling parameter m,global_threshold and local_threshold
global_search_time = 0, local_search_time =0,
cost(C_curBest) = MAX_VALUE
while global_search_time < global_threshold do
    global_search_time++, local_search_time = 0
    samples[m] = conditional_LHS(m)
    ConfigList = satisfyConstraint(samples[m])
        if configList is empty
            continue
        end if
    ConfigResult = assignToTasks(ConfigList)
    updateConstraintsAndRanges(ConfigResult)
    if cost(best(ConfigResult)) < cost(C_curBest)
        C_curBest = best(ConfigResult)
    else
        continue
    end if
    while local_search_time < local_threshold do
        local_search_time++
        NeighbourList = getNeighbours(C_curBest)
        ConfigList = satisfyConstraint(NeighbourList)
        if configList is empty
            break
        end if
        ConfigResult = assignToTasks(ConfigList)
        updateConstraintAndRanges(ConfigResult)
        if cost(best(ConfigResult)) < cost(C_curBest)
            C_curBest = best(ConfigResult)
        else
            continue
        end if
    end while
end while
```

Searching Optimal Configurations

1) Dividing Parameters Search Space
--map-relevant parameters
--reduce-relevant parameters

2) Searching Algorithm
--propose model-based hill climbing algorithm
-- The global search aims at covering the search space as broader as possible
--The local search phase starts from current best configuration *C_curBest* and seeks the steepest direction by exploring *C_curBest*'s neighborhood

3) Parameter Constraints
--parameter model emphasis on how the selected parameters impact on the cost of a task

--During the execution of a job, the system continually collect the size and record numbers of map output as *MapOutputByte* and *MapOutputRec* in each task and keep the greatest one in *MaxMapOutputByte* and *MaxMapOutputRec*

--In global search, use the updated values to narrowing the scope of *PsortFactor*, *PmapBuf*, *PredJavaOpts*

4)Configuration Evaluation
--Evaluation functions are defined to assess configurations: map-relevant configuration, reduce-relevant configuration

--add the ratio of spill records to eliminate impacts of data skew

--The ratio of Java heap size is also included to avoid over-allocation of buffer

Online-Tuning Approach

--relies on the theory that all the map/reduce tasks have the same execution logic and similar input data in a job

-- tasks run in multiple waves

--assign different configurations to the tasks and collect their running statistics

--evaluate these configurations with their statistics and get a suitable one for the job

Elastic Container

---the size of an elastic container can temporarily expand by taking idle resources from other containers or the node, and shrink by handing resources back to the node

Resource Re-Scheduling

Rescheduling Strategies

---goal: improving the parallelism of a job and making full use of resources in the node, arrange containers as much as possible in each node

---In previous tuning step, get the memory usage of a map/reduce task with the optimized configuration

---adjust memory allocation per container according to the maximum memory usage during task execution

--When *resource rescheduler* detects that a container needs more resources to maintain task execution or improve task performance, it assigns idle resources to this container temporarily

--When borrowed resources have to return to the system, the container gives these resources back

--use scaling = needs + unfairness to evaluate which container has the biggest chance to obtain extra resources
A positive value of *needs* means that a container is in a resource-critical situation and needs more resources
A positive value of *unfairness* suggests that a container has lent resources from others

Resource Re-Scheduling

-- when detect full CPU utilization in a container, algorithm add idle CPU resources to it.

--If adding CPU resources has no performance improvement and CPU usage of the container is at a low level, it retrieves the CPU resource back to the idle resource pool

Evaluation

Benchmark	Input Data	Input Size	# Maps	#Reduces	Label
TeraSort	TeraGen	50 GB	374	99	J1
WordCount	Wikipedia	50 GB	396	99	J2
Grep	Wikipedia	50 GB	396	99	J3
Inverted Index	Wikipedia	50 GB	396	99	J4
Classification	Movie ratings dataset	50 GB	422	0	J5
HistogramMovies	Movie ratings dataset	50 GB	422	99	J6
HistogramRatings	Movie ratings dataset	50 GB	422	99	J7

Effectiveness of Configuration Tuning Process

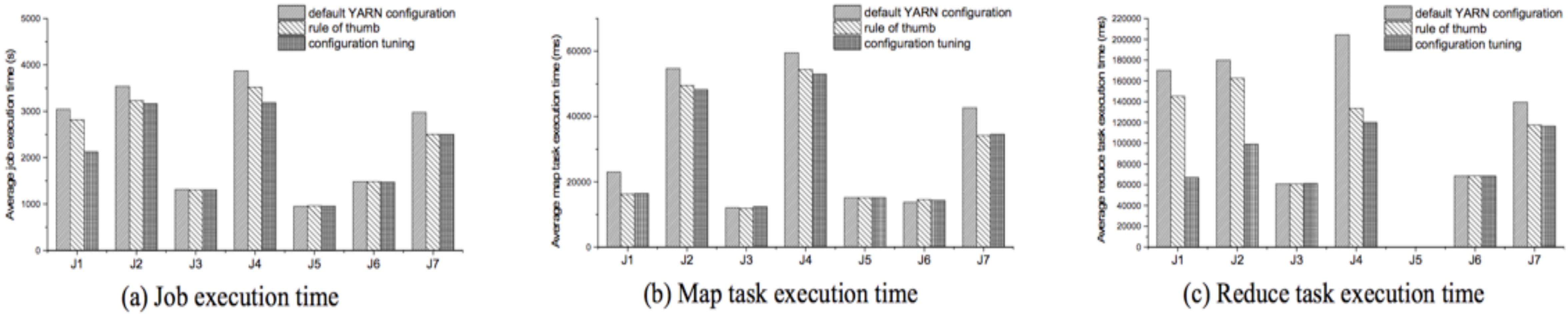


Fig. 3. Effectiveness of Configuration Tuning Process

--(a) compares average job completion time using default configuration, rule of thumbs , and configuration tuning process.
--(b), (c) show average execution time of reduce tasks has more improvement than map tasks
Reason: they change the value of *input.buffer.percent*, which decides the maximum records a reduce task can retain in the buffer.   The default value of this parameter is 0

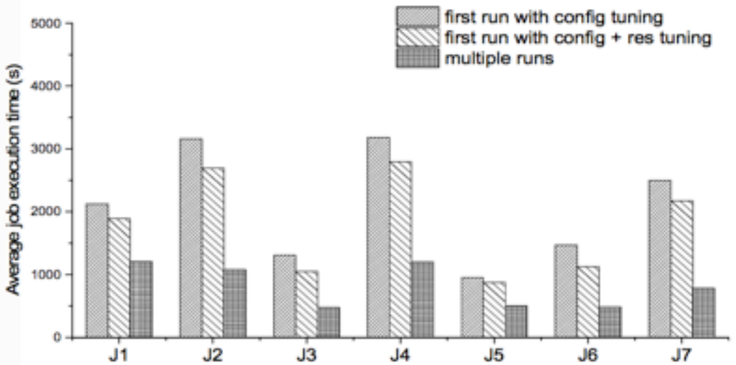
Evaluation

(a) Spill records of J1, J2, J4

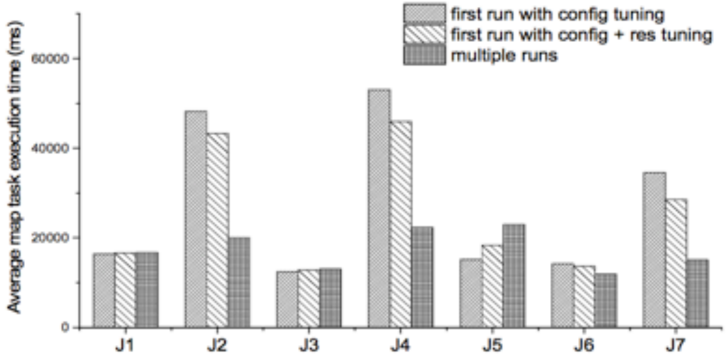
(b) Spill records of J3, J5, J6, J7

Fig. 4. Spill records of a job with default YARN configuration, rule of thumbs, and configuration tuning process in JellyFish.

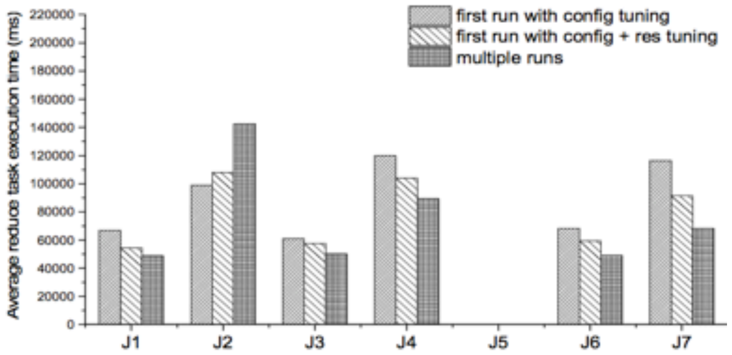
--the reduction in spill records accounts for the reduction in execution time.
--J3,J5,J6 has relatively fewer map output and shuffle records, thus fewer spill records
--This configuration tuning process has less effects on them



(a) Job execution time



(b) Map task execution time



(c) Reduce task execution time

-- evaluate the effectiveness of resource rescheduling process

-- The first case is that a job starts with the default configuration and runs with configuration tuning process.

-- The second is that a job starts with default configuration and runs with both two tuning processes.

--The third is that a job begins with a suitable configuration obtained in previous job execution and runs with resource re-scheduling process

-- (a) compares various completion time in different case

-- (b), (c) illustrate that JellyFish shortens task execution time with resource rescheduling for most jobs

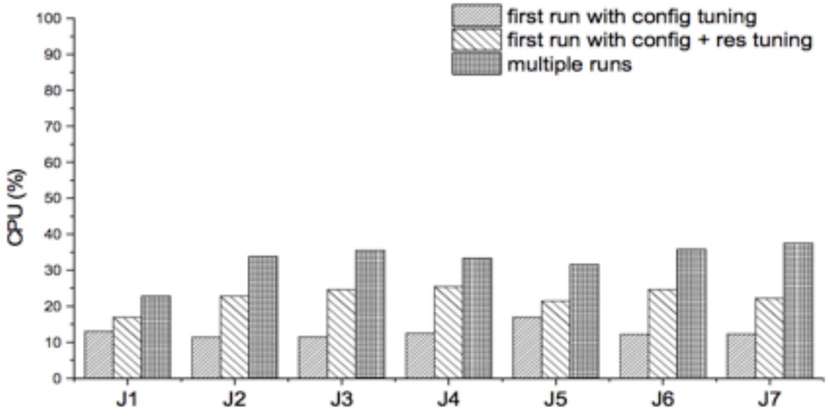
-- tasks in J2 and J5 have longer execution time because both of them are CPU-intensive applications. They are in CPU-contention when adding more containers

Evaluation

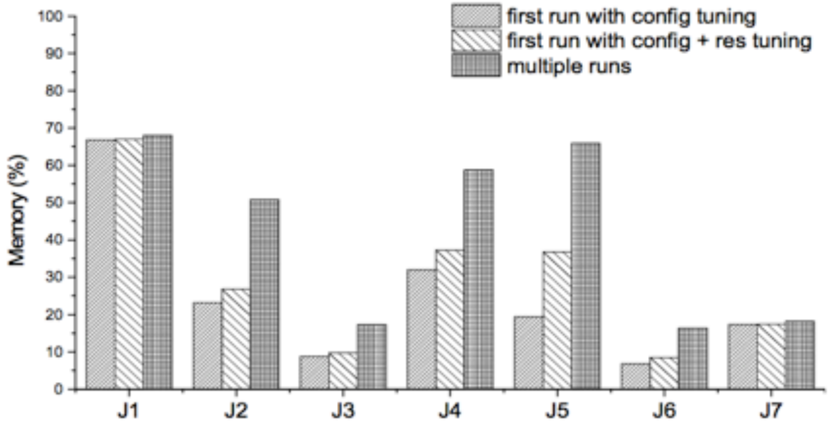
Application	first run with config tuning	first run with config + res tuning	multiple runs
J1	30%	38%	61%
J2	11%	24%	70%
J3	1%	20%	64%
J4	18%	28%	69%
J5	0%	8%	48%
J6	1%	24%	68%
J7	16%	27%	74%

---performance improvement by comparing execution time with YARN default configuration

Evaluation

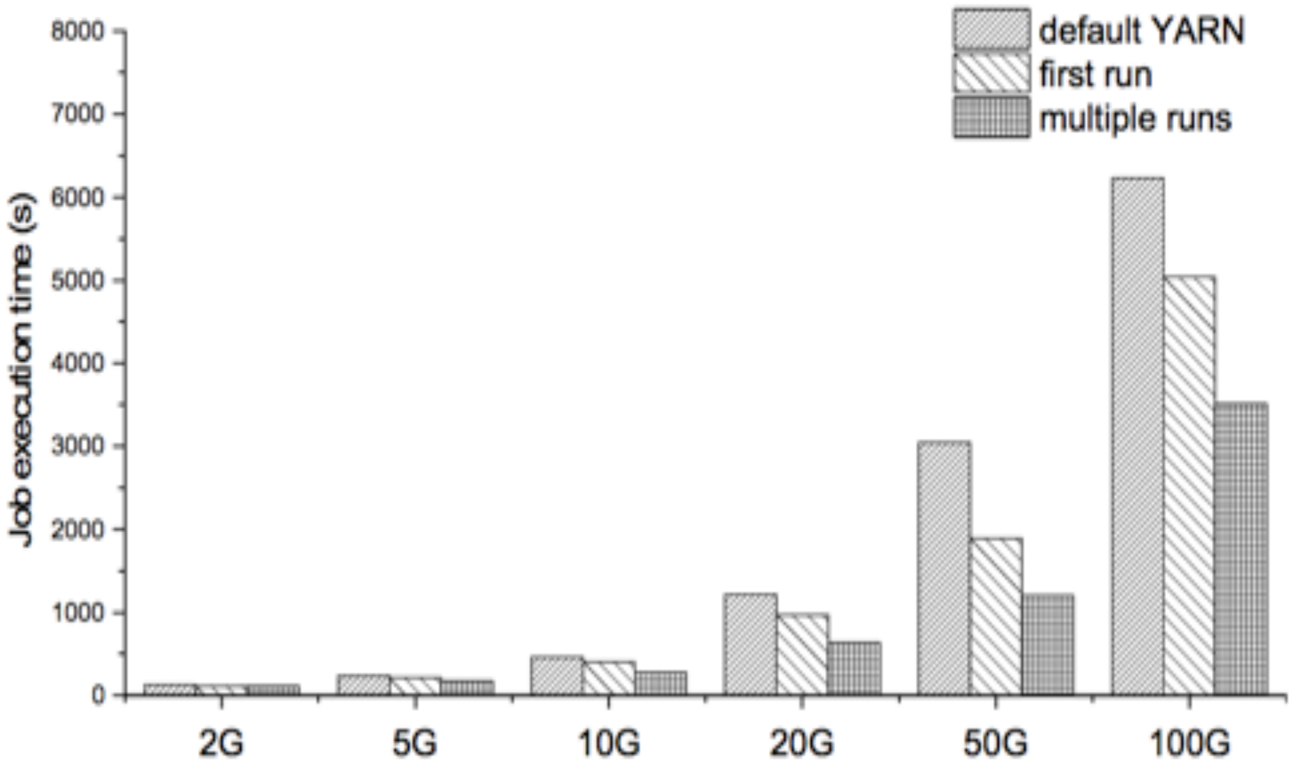


(a) CPU utilization in the cluster.



(b) Memory utilization in the cluster

-- Improvement of resource utilization in the cluster
-- (a) illustrates that CPU utilization has been tripled
-- (b) reveals that memory resource utilization also increases
--J1 and J7 have less improvement because their memory usage is almost the same with resource allocation in default



--the effectiveness of JellyFish with different job size

--run Terasort with increasing input data sets ranging from 2GB to 100GB

-- JellyFish can reduce job execution time significantly in jobs run more than once

--Jobs run for the first time in JellyFish have relatively longer execution time because it spends most of time in searching for the desirable configuration