2012 IEEE 4th International Conference on Cloud Computing Technology and Science

Predator - An Experience Guided Configuration Optimizer for Hadoop MapReduce

Kewen Wang, Xuelian Lin, Wenzhong Tang
School of Computer Science and Engineering
Beihang University
Beijing, China
wkw@cse.buaa.edu.cn，{linxl, tangwenzhong}@buaa.edu.cn

Abstract — **MapReduce is a distributed computing programming framework which provides an effective solution to the data processing challenge. As an open-source implementation of MapReduce, Hadoop has been widely used in practice. The performance of Hadoop MapReduce heavily depends on its configuration settings, so tuning these configuration parameters could be an effective way to improve its performance. However, picking out the optimal configuration settings is not easy for the time consuming nature of MapReduce together with the high dimensional and nonlinear features of its configuration optimization. In this paper, we introduce Predator, an experience guided configuration optimizer, which does not treat the optimization problem as a pure black-box problem but utilizes useful experience learnt from Hadoop MapReduce configuration practice to assist the optimizing process. The optimizer uses job execution time estimated by a practical MapReduce cost model as the objective function, and classifies Hadoop MapReduce parameters into different groups by their different tunable levels to shrink search space. Furthermore, the optimization algorithm of the optimizer uses the idea of subspace division to prevent local optimum problem, and it could also reduce the searching time by cutting down the cost in visiting unpromising points in search space. Experiments on Hadoop clusters demonstrate the effectiveness and efficiency of the optimizer.**

Keywords-MapReduce, Hadoop ,Configuration, Optimization

I. INTRODUCTION

MapReduce [1] is a distributed computing programming framework which provides an effective solution to the data processing challenge. As an open-source implementation of MapReduce, Hadoop [2] has been widely used in practice, especially for its easy deployment and open source feature. Many companies have used Hadoop for data mining, user behavior analysis and scientific simulations. To make full use of cluster resources including CPU, memory and I/O, it's important to optimize Hadoop performance according to specific applications. Thus issues about Hadoop performance improvement have been becoming the concern of application developers, some of whom modify Hadoop core codes to improve its performance.

Although rewriting the Hadoop codes is the essential method to make Hadoop adapted to its different applications, it should not be the general way of optimizing MapReduce programs because it is time consuming. Fortunately, tuning Hadoop configuration parameters provides a better way out. Hadoop has more than 200 configuration parameters, which decide the overall performance of Hadoop MapReduce. Among these parameters, about 20 of them significantly affect MapReduce job performance. Tuning these parameters is beneficial to Hadoop because appropriate configuration settings can shorten the job execution time, increase throughput and reduce I/O or network transmission cost [3].

However, facing so many configuration parameters, setting proper parameter values to optimize job performance is tough even to experienced Hadoop developers, who usually tune each parameter according to their experience about the parameter's impact on MapReduce job. Besides, most parameters have a side effect on job performance, which means that changing the value of this parameter can reduce one cost, but might increase other costs as well. For example, setting the parameter *mapred.compress.map.output* to be true (default is false) can reduce the I/O and network transmission cost by decreasing intermediate data size transferred from Mappers to Reducers, but add CPU overhead of the compression and decompression processes. Moreover, some parameters correlate with each other, the map-side sorting buffer size is determined by *io.sort.mb* whose upper bound is less than JVM heap size which is set by *mapred.child.java.opts*. Therefore, there is a need to tune configuration parameters in a generic way.

Current method [4] to solve this problem is to first establish a parameter space including all possible values for each parameter, then to use search algorithm to find the optimal parameter settings given the objective function. This method treats Hadoop configuration optimization problem as a black-box optimization problem [5]. But the objective function of this configuration optimization problem is usually high dimensional and nonlinear, so using pure black-box optimization method to pick out globally optimal parameter settings is time consuming and inefficient. In addition, searching through the whole parameter space without clear direction or suggestion except the objective function appears neither effective nor efficient.

This paper presents Predator - a Hadoop configuration optimizer, which combines the parameter tuning experience with search algorithm. The knowledge of configuration tuning experiences provides some suggestions during the

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

search process. For example, according to this experience, *mapred.tasktracker.map* and *reduce.tasks.maximum* all should be set to be a value between (cpu_cores_per_node)/2 and $2 \times$ (cpu_cores_per_node) [6], which significantly decreases the value range of the parameter. Discovering patterns from the experiences could provide suggestions about parameter settings while searching the parameter space, and help shrink the search space through setting specific value or narrowing the value range for parameters to finally improve the searching efficiency.

Motivated by these facts, this paper establishes a Hadoop configuration model to classify these parameters into four groups. The first group of parameters is merely determined by general configuration experience; the second group of parameters could be adjusted given further information about cluster's CPU and memory; the third group of parameters requires job input information to be adjusted; the fourth group of parameters that have more relations with each other than the other three groups, should be left to the search algorithm. And then each group of parameters is preprocessed according to the features of the specific group.

Besides, this optimizer uses the estimated job execution time as the objective function, which is applied in our search algorithm. First we gather the primary information about the previous job from its history logs and profiling files that could be obtained by applying BTrace [7]. Then by analyzing this information, we extract the job properties relevant to its performance. Based on these job properties and our cost model, we construct the hypothetical job with different configurations. And the execution time of the hypothetical job can be obtained as the objective function.

Moreover, this paper proposes a Grid Hill Climbing algorithm (GHC) to search for the optimal configuration. Because Hadoop has about 20 parameters to be optimized, we need to search in a high dimensional space. The feature of high time-complexity makes the deterministic algorithms like branch-and-bound or dynamic programming not suitable under this circumstance. Since its objective function is not definite, it's impossible to describe the objective function in the form of exact equations. Direct search methods like Newton's methods, steepest decent are also not appropriate. Current methods on this high dimensional optimization utilize heuristic search algorithms [8]. But heuristic search algorithms usually have the local optimization problem. To solve this problem, our algorithm randomly chooses several most promising points in equally divided parameter subspaces. Assisted by the general configuration experience, Grid Hill Climbing algorithm is more efficient than pure random search algorithm without definite guidance.

The rest of the paper is organized as follows. Section II introduces the related work. Section III gives the overview of Predator. Section IV describes the configuration model in detail and displays the process of adjusting these parameters in each group. Section V describes the objective function applied in our search algorithm. Section VI discusses the existing algorithm solving this problem and provides the details of Grid Hill Climbing algorithm. Section VII presents experimental results about Predator. Finally we conclude this paper in Section VIII.

II. Related Work

The main idea about configuration optimization is to apply a search algorithm to search through the whole parameter space according to an objective function, and find the optimal configuration. These methods always treat this optimization problem as a pure black-box optimization problem [9].

In distributed systems, an evolutionary algorithm called Covariance Matrix Adaptation (CMA) [10] is used to search for optimal settings and automatically tune configuration in distributed systems. But the objective function value is obtained through real system's running, which takes as many as 25 minutes. It is time consuming to evaluate objective function, and limits the number of evaluation. In the context of Internet services automatic configuration, a parameter dependency graph [11] is used in searching algorithm to reduce the searching times, but this method needs several times of service exercising, which is also a high cost.

To solve large-scale network parameter configuration problem, an on-line simulation framework [12] is used to model the network conditions and estimate the job execution with different settings. It could reduce the cost of objective function evaluation without running on real network environment. Besides, in this context, recursive random search algorithm [13] is applied in the searching process. For the random sampling feature, it is quick in searching, but this is also its drawback that has no clear direction.

There is not so much research in Hadoop configuration optimization. Starfish [4] is an attempt in this field. The starfish system first collects the previous MapReduce job's profile information by BTrace [7], then estimates the virtual job's running time through simulating this job's execution with different configurations based on the job's profile information and a What-If Engine, and searches through the parameter space to find the optimal configuration settings with shortest estimated job execution time. Simulating job's execution is a better approach to evaluate objective function with less cost compared with the approach to run real job. But the search strategy adopted in starfish is recursive random search algorithm, whose random feature makes it inefficient in obtaining global optimum.

Practical experience about Hadoop configuration is an important source of improvement in Hadoop configuration optimization. An approach in Hadoop performance tuning methodologies and best practices [14] provides a good example. It discusses several tuning techniques involving Hadoop, JVM, OS and BIOS configuration parameters tuning. Although it is a demo paper, it provides guidelines for Hadoop parameters configuration, especially for those parameters relevant to JVM, memory and disk. Corporations like Intel, IBM and Impetus also give empirical suggestions about Hadoop configuration in their white papers [6,15,16].

Experience Transfer [17] is an idea to utilize the configuration knowledge learnt from previous system to benefit the configuration in another similar system. This strategy emphasizes the experience transfer between similar computing systems, and use Bayesian network to apply the previous experience in current system.

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

Our work is different from all the work above. Guided by the practical experience about Hadoop configuration, we divide these parameters into separate groups, and use our search algorithm to find the best parameter settings with shortest estimated MapReduce job execution time, which is the objective function based on our cost model.

III. Overview of Predator

Hadoop configuration optimization problem is a high dimensional optimization problem. But it is not a pure black-box problem because we can obtain suggestions by learning practical experience about parameter tuning. Such experience provides useful information about parameter values or value range of the optimal configuration. Directed by this experience, optimizing process will be more efficient.

Predator selects 23 parameters that have the most significant impact on the performance of MapReduce job. The basic idea of Predator is to preprocess these parameters according to the configuration model and then use experience-combined search algorithm to find the optimal configuration based on the objective function. The execution overview of Predator is displayed in Fig.1 involving the following steps.

1) Establish a configuration model to classify these parameters into four groups according to their tunable levels on the basis of configuration experience.
2) Preprocess every parameter on each group by setting specific value or narrow value range for parameters.
3) Gather previous job's execution information and analyze this information to obtain the job properties relevant to its performance.
4) Based on our cost model and these job properties, construct the hypothetical jobs with different configurations for the points in the parameter space, and obtain each job's execution time as the objective function.
5) Divide the parameter space into equal subspaces.
6) Exploit Grid Hill Climbing algorithm (GHC) to search for optimal configuration settings on the basis of our objective function.

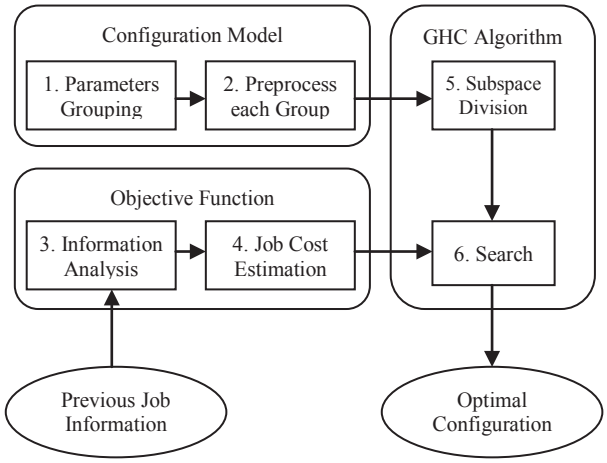


Figure 1. The Execution Overview of Predator

During the execution of Predator, the configuration knowledge from previous experience is not only limited in step 2) as a prior knowledge to filter out unpromising points in the parameter space, but is also used in step 6) as an advice to guide searching.

IV. Configuration Model

Learning from the previous experience about Hadoop configuration, we find that some suggestions about particular parameters could help us directly determine their optimal values while some other parameters need extra information about cluster configuration or job inputs to tune their values, leaving a few parameters without change. Basically, these unchanged parameters also have been optimized as their relevant parameters are adjusted. This configuration model classifies parameters into the following four groups.

A. Group 1 - Determinable Parameters

Parameters in the first group are merely determined by the previous experience because many corporations suggest that these parameters should be set to these values that have positive impact on job performance. For example, using LZO compression codec could significantly improve job performance, which is demonstrated by Intel experiment [6]. Parameters of this group are listed below in Table I which also gives the suggested values.

B. Group 2 - Cluster Relevant Parameters

Parameters in the second group could be adjusted according to the information of processor and memory in the cluster. Parameters of this group are listed in Table II.

In this group, the first and second parameters determine the maximum number of Map or Reduce tasks that can run simultaneously on one node (also the number of Map or Reduce slots). The sum of task slots should be no more than $2\times$(cpu_cores_per_node) if each core of CPU supports simultaneous multi-threading. So the number of Map or Reduce slots should be between (cpu_cores_per_node)/2 and $2\times$(cpu_cores_per_node) [6]. Generally, the two parameters should be set to the same value: cpu_cores_per_node.

The third parameter determines the maximum JVM heap size allocated to each task. If each work node has both Datanode and TaskTracker daemons, and each daemon costs 1 GB memory [15], the memory left should be allocated to Map or Reduce tasks running simultaneously. Usually, this parameter is set to 80% of the available memory. But if the total memory of each node is no more than 2 GB, this parameter should be set to the default value.

The last parameter *mapred.reduce.tasks* is the number of Reduce tasks for a job, which should be adjusted in the range of $(0.95\sim1.75)\times$ *mapred.tasktracker.reduce.tasks.maximum*, and we should avoid multiple Reduce waves which is defined as (the number of Reduce tasks) / (the number of Reduce slots), because more Reduce waves could add extra shuffle latency [18]. Thus this parameter is often set to the number of Reduce slots.

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

C. Group 3 - Input Relevant Parameters

To set the value of parameters in the third group, we need information on job input, and parameters of this group are listed in Table III.

The first parameter denotes the minimum size of Map input split, which indirectly determines the number of Map tasks because each Map task is allocated a split. Given the number of Map slots, we should adjust this parameter to generate multiple Map waves in order to hide shuffle latency [18], where the Map wave is defined as (the number of Map tasks) / (the number of Map slots). Besides, the right number of Map tasks per node is around 10~100. Thus we should adjust this parameter value according to these principles.

The second and third parameters denote whether we should turn on speculative execution for Map or Reduce task. Setting these parameters to be true could decrease some tasks' execution time by killing slow tasks, but it also reduce the overall throughput on a busy system for redundant tasks are executed. So for large job whose tasks' average execution time is significant (more than an hour) and the overall throughput of this system is high, these two parameters should be set to false [16].

The fourth parameter specifies the number of tasks per JVM. And the overhead of starting JVM for every task is around one second. For the tasks whose execution time is about seconds or a few minutes, setting this value to the number of total tasks (Map and Reduce tasks) can save much time for the job.

D. Group 4 - Undeterminable Parameters

Parameters in the fourth group could not be directly changed from the experience. But these parameters could be optimized because their relevant parameters are adjusted in other groups. Moreover, tuning experience also gives suggestions to adjust these parameters. Parameters of this group are listed in Table IV.

TABLE I. DETERMINABLE PARAMETERS

No.	Name	Default Value	Suggested
1	mapred.output.compress	fale	true
2	mapred.output.compression.type	RECORD	BLOCK
3	mapred.output.compression.codec	DefaultCodec	LZO
4	mapred.compress.map.out	false	True
5	mapred.map.output.compression.codec	DefaultCodec	LZO

TABLE II. CLUSTER RELEVANT PARAMETERS

No.	Name	Default Value	Suggested
1	mapred.tasktracker.map.tasks.maximum	2	cores/2~2×cores
2	mapred.tasktracker.reduce.tasks.maximum	2	cores/2~2×cores
3	mapred.child.java.opts	-Xmx200m	80%×Mem
4	mapred.reduce.tasks	1	(0.95~1.75) ×Max

TABLE III. INPUT RELEVANT PARAMETERS

No.	Name	Default Value
1	mapred.min.split.size	0
2	mapred.map.tasks.speculative.execution	True
3	mapred.reduce.tasks.speculative.execution	True
4	mapred.job.reuse.jvm.num.tasks	1

TABLE IV. UNDETERMINABLE PARAMETERS

No.	Name	Default Value
1	io.sort.mb	100
2	io.sort.factor	10
3	io.sort.record.percent	0.05
4	io.sort.spill.percent	0.8
5	min.num.spill.for.combine	3
6	mapred.job.shuffle.input.buffer.percent	0.7
7	mapred.job.shuffle.merge.percent	0.66
8	mapred.inmem.merge.threshold	1000
9	mapred.job.reduce.input.buffer.percent	0.0
10	mapred.reduce.slowstart.completed.maps	0.05

The parameter *io.sort.mb* denotes the sorting buffer size on the Map side. Increasing this value will generate fewer spills to the disk, reducing I/O times as a result, but it adds the memory requirement of each Map task. So we should set this value according to the available memory. In general, this value should be set to about 70% of maximum JVM heap size that is specified by *mapred.child.java.opts*. The second parameter specifies the number of streams to merge at once while sorting files on both Map and Reduce sides. The greater value it is, the fewer spills and less I/O times there will be. In addition, we should set this parameter to large enough to fully utilize the sorting buffer size specified by *io.sort.mb*. The following parameter *io.sort.record.percent* determines the percent of *io.sort.mb* dedicated to tracking record boundaries. The next parameter *io.sort.spill.percent* is the threshold for sorting and record buffer. When this percentage of either buffer has filled, their contents will be spilled to disk in the background. The parameter *min.num.spill.for.combine* is the minimum number of spill files needed for the combiner to run. When it is set to 3, combiner will run when there are at least 3 spill files. Setting this parameter to a large value will reduce data size written to the disk and decrease I/O times as a result.

The parameter *mapred.job.shuffle.input.buffer.percent* represents the percent of maximum JVM heap size that can be allocated to store Map outputs during the shuffle. Setting it to the default value (0.7) is sufficient for most job, but to large job with significant Map outputs, we should set it to 0.8~0.9. Next parameter *mapred.job.shuffle.merge.percent* determines the memory threshold at which an in-memory merge is started, expressed as a percentage of memory allocated to store Map outputs in memory. And it should be set to 0.8(more than the default value) according to the experience. The parameter *mapred.inmem.merge.threshold* is

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

the number of fetched Map outputs in memory before being merged to disk. And it should be set to a large value when Map tasks generate many lightweight output files. The parameter *mapred.job.reduce.input.buffer.percent* denotes the percent of maximum JVM heap size to retain Map output in memory during Reduce. The greater value this parameter is, the less memory available for Reduce there will be. Thus, for the job which consumes much memory during Reduce, this parameter should be set to a small value or 0 by default. The last parameter is the fraction of the number of Map tasks that should be complete before Reduce tasks are scheduled for the job. This parameter has significant impact on Map and Reduce tasks. If the shuffle phase of Reduce takes long, we should set this parameter to a small value to hide shuffle latency, but it will result in task slot occupied by Reduce very early. If Map tasks' execution time is much longer than Reduce tasks, this parameter could be set to a larger value to hide shuffle latency.

E. Analysis of Configuration Model

After preprocessing these parameters based on the configuration model, the search space largely shrinks. The reduction in search space could be demonstrated by the analysis below.

The number of parameters to be optimized is n, each has K_i $(0<i<=n)$ values in its range, so the dimension of this search space is n, and the total number of points in the space can be defined as $\text{Num} = \prod_{i=1}^{n} K_i$.

After optimized by the configuration model, c parameters among them are fixed to a specific value, and d parameters' value range shrink to r_j $(r<1,\ c<j<=c+d)$ of the original range, left other $n-c-d$ parameters without change. The total number of the points after optimization is calculated by $\text{Num}_s = \prod_{i=c+1}^{c+d}(r_i \times K_i) \times \prod_{j=c+d+1}^{n} K_j$.

Thus the ratio of improved search space points' number to the original search space points' number is Num_s/Num, suppose $R = \text{Num}_s/\text{Num}$, $R = \prod_{i=1}^{c}(1/K_i) \times \prod_{j=c+1}^{c+d} r_j$.

In our configuration model, the total number of parameters is $n = 23$. The parameters in the first group are set to a fixed value, which is selected from two alternatives, and $c = 5$. In the second and the third groups, the parameters' range is narrowed to a smaller one, which is less than the half of the original range, and $d = 8$. Therefore, $R < 1/2^{13}$, which is approximately to 0.01%. That's a significant decrement in search space.

V. Model-based Objective Function

Before applying searching algorithm, we should first define the objective function. The objective function of our algorithm is defined as the estimated execution time of a hypothetical job that simulates how the job will execute with a different configuration generated by the point in parameter space.

A. Initialization

We first gather primary information of the previous MapReduce job from its history logs and profiling files that could be obtained by a dynamic tracing tool BTrace [7]. Then analyze the information to extract job's performance relevant properties such as relations between each phase's execution time and input data size, dataflow statistics [19] and so on.

B. Cost Model

According to these important job properties, we could estimate each task's execution time of hypothetical job based on our cost model [20] which defines Map cost and Reduce cost as vectors.

Map cost vector is defined as:

$$T_{map} = (T_{m1\_init}, T_{m2\_read}, T_{m3\_net}, T_{m4\_parse}, T_{m5\_mapper}, T_{m6\_sort}, T_{m7\_merge}, T_{m8\_serial}, T_{m9\_read}, T_{m10\_write}). \quad (1)$$

Reduce cost vector is defined as:

$$T_{reduce} = (T_{r1\_init}, T_{r2\_read}, T_{r3\_net}, T_{r4\_merge}, T_{r5\_serial}, T_{r6\_io}, T_{r7\_parse}, T_{r8\_reducer}, T_{r9\_net}, T_{r10\_write}). \quad (2)$$

Because some sub phases of a task are parallel, the execution time of one Map task is estimated by:

$$T_{Map} = T_{m1\_init} + \text{Max}\{T_{m2\_read}, T_{m3\_net}, (T_{m4\_parse} + T_{m5\_mapper})\} + T_{m6\_sort} + \text{Max}\{T_{m7\_merge}, T_{m9\_read}\} + \text{Max}\{T_{m8\_serial}, T_{m10\_write}\}. \quad (3)$$

And the execution time of a Reduce task is estimated by:

$$T_{Reduce} = T_{r1\_init} + \text{Max}\{(T_{r2\_read} / \text{Min}\{\text{pCopy}, \text{nMap}\}), T_{r3\_net}\} + \text{Max}\{(T_{r4\_merge} + T_{r5\_serial}), T_{r6\_io}\} + \text{Max}\{(T_{r7\_parse} + T_{r8\_reducer}), T_{r9\_net}, T_{r10\_write} / \text{dRep}\}. \quad (4)$$

Where pCopy is *mapred.reduce.parallel.copies*, nMap is *mapred.map.tasks*, and dRep is *dfs.replication*.

C. Objective Function

Based on this cost model, we could estimate Map and Reduce execution time separately by:

$$\text{Time}_{Map} = \frac{\text{nMap} \times T_{Map}}{\text{nNode} \times \text{sM}}. \quad (5)$$

$$\text{Time}_{Reduce} = \frac{\text{nReduce} \times T_{Reduce}}{\text{nNode} \times \text{sR}}. \quad (6)$$

Where nNode is the number of work nodes in the cluster, nReduce is *mapred.reduce.tasks*, sM is Map slots' number: *mapred.tasktracker.map.tasks.maximum*, and sR is reduce slots' number: *mapred.tasktracker.map.tasks.maximum*.

MapReduce job consists of four phases: Setup, Map, Reduce, Cleanup, where Setup's execution time Time_{Setup} and Cleanup's execution time $\text{Time}_{Cleanup}$ are constant time. Thus we could get hypothetical job's execution time, which is also the objective function F by:

$$F = \text{Time}_{Setup} + \text{Time}_{Map} + \text{Time}_{Reduce} + \text{Time}_{Cleanup}. \quad (7)$$

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

VI. GRID HILL CLIMBING ALGORITHM

For the high dimensional feature of this optimization problem, current approach [13] is to use heuristic search algorithm. But heuristic algorithms like Hill Climbing [21] cannot ensure that the local optimum is approximately the global optimal solution. In this paper, we propose a Grid Hill Climbing algorithm to solve this problem by randomly selecting several promising points in equally divided subspaces. In this algorithm, we use the practical experience in searching process.

The Grid Hill Climbing algorithm is described below.

1) Initialize parameters space by modifying some parameters' range and value according to the configuration model based on the previous experience.

2) For each parameter, divide the parameter range into C non-overlapping intervals with equal possibility. For d parameters, the space is divided into m ($m = C^d$) subspaces.

3) Randomly generate a sampling point in each parameter subspace, and choose n ($n < C^d$) candidate points p_i ($0<i<=n$) with the least value of F (objective function) from these m samples.

4) For each candidate point p_i ($0<i<n$), use hill climbing search to find the best point bp_i within the neighborhood range of p_i .

a) Initialize attempt times t, select p_i as the center point cp of local search, and set the best point $bp_i = p_i$

b) For each attempt j ($0<j<t$), find k neighbors of the center point cp.

i. If the F values of these neighbors are all larger than the center point's , set the best point $bp_i = cp$, go to step 4) to begin next search.

ii. Else obtain the minimum point mp with minimum F value among k neighbor points and the center point. Update $cp = mp$, and set the best point $bp_i = mp$, go to step b) to start next attempt.

5) Select the point with minimum F value as the optimal point op among bp_i (0<i<n).

6) Generate the configuration of op as the optimal configuration.

Generally, sampling is important to heuristic search algorithms. Hill climbing is efficient in reaching the local optimum, but it easily falls into the local optimization. The reason is that the random sampling could not guarantee the local optimum is also global. Recursive random search [13] also has this problem, since it makes use of repeat random sampling to generate start point and neighbor points. This process may cost extra time in generating points without getting progress in approaching the promising points or even optimal point in global.

Our method is to divide the whole searching space into equal subspaces, and randomly choose one point in every subspace, so n promising points of which are saved as the center points that are used later to generate neighbor points in hill climbing search phase. This technique could ensure the sampling points at the beginning are global.

In the hill climbing search phase, choosing a good neighbor searching method is also essential to the efficiency of the algorithm. In this algorithm, we define the neighbor as follows:

Every point has d dimensions, each of which is the value of one parameter. And every center point has at most $2 \times d$ neighbors, which is only different from center point in one parameter value (smaller or larger). The distance between each value is one interval equally divided in the value range. For example, for the points with 2 parameters, suppose point A=(1,1), the neighbors of A is (1,0), (1,2), (0,1), (2,1) if the interval is 1 and parameter value is in the range of (0~2). Besides, this interval is often fine-grained compared with the interval mentioned in step 2) of our algorithm.

Moreover, this algorithm also takes practical experience as an advice during searching process by adjusting the parameter value's interval. For example, the number of Map or reduce tasks should be set to multiple of respective task slots' number to fully utilize cluster resources. So these parameters' values should be changed every certain value, which is equal to the number of task slots. General experience also points out the search direction for parameters. For example, setting *io.sort.factor* to a larger value is better than setting it to a smaller one.

VII. EXPERIMENTAL RESULTS

A. Experimental Setup

The experiment is made on a Hadoop cluster of five nodes running MapReduce job. The physical configuration of these five nodes is listed below in Table V.

The whole experiment consists of two parts. The first part is to evaluate the effectiveness of Predator based on the Configuration Model (CM) and Grid Hill Climbing (GHC) algorithm. The second part is to evaluate the efficiency of this optimizer.

B. The Effectiveness

In this part, we compare the MapReduce job's execution time using default configuration settings, the configuration settings suggested by optimization based on Random Recursive search (RRS) algorithm and Predator based on the Configuration Model and Grid Hill Climbing algorithm (GHC-CM).

To show the performance of the job running, we provide the minimum, mean and maximum values of each set of test, which is run at least 10 times. Fig.2 uses WordCount MapReduce program to test the performance of each job with different configurations. It lists the real job's execution time with 1G, 5G, 10G input data (documents from Wikipedia). The results show that the configuration suggested by Predator is better than default settings and configuration suggested by RRS-based optimization. Besides, even the maximum execution time with configuration suggested by Predator is less than the minimum execution of the others, which could further demonstrate the effectiveness of Predator.

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

When comparing the results of RRS and Predator, we could find the clear advantage of Predator. The improvement of Predator over RRS-based optimization is more obvious in Fig.2(c), and in this test, the job performance is even worse after optimized by RRS. The result is partly from the LZO compressor which is enabled in configuration settings suggested by Predator because compression reduces the data size transferred between these nodes and finally decreases the I/O cost.

In Fig.2, we also list the experimental results of RRS combined with our Configuration Model (RRS-CM). The comparative results of RRS and RRS-CM demonstrate the power of our configuration model, which preprocesses the configuration parameters. In Fig.2(c), the improvement of RRS-CM over RRS is partly attributed to the intermediate data compression, which is enabled by CM. Besides, the gap between RRS-CM and Predator (GHC-CM) is also a proof of our GHC algorithm's effectiveness in finding better optimal configuration settings.

In Fig.3, we repeat the experiment of Fig.2 but use a different program TeraSort (input data is generated by Hadoop's TeraGen) to demonstrate the effectiveness of Predator. In this standard benchmark, Predator shows clearer advantage over RRS-based optimization. Predator achieves more than 68% improvement over RRS-based optimization in Fig.3(b), and this improvement rises to more than 88% in Fig.3(c). More obvious advantage of our GHC algorithm compared to the experiments conducted in Fig.2 results from that TeraSort job stresses more on the balance between computing and data I/O, which is also Predator's emphasis. Besides, the gap between RRS-CM and GHC is larger than the gap in Fig.2, which further proves the effectiveness of GHC.

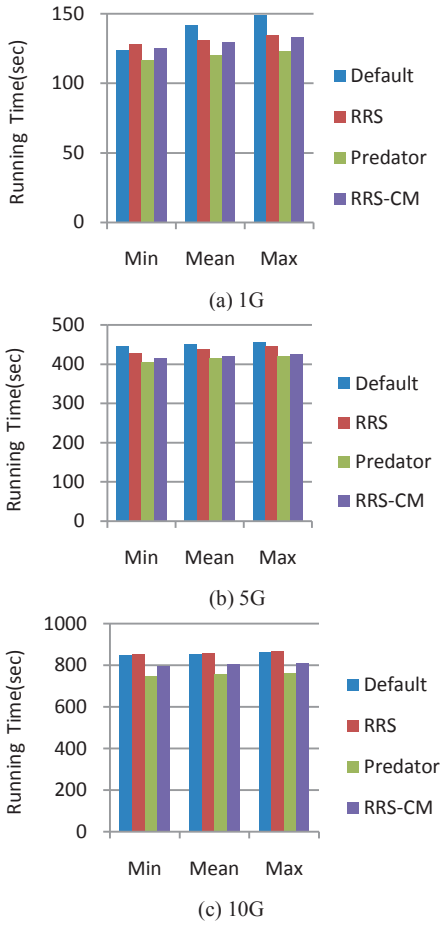


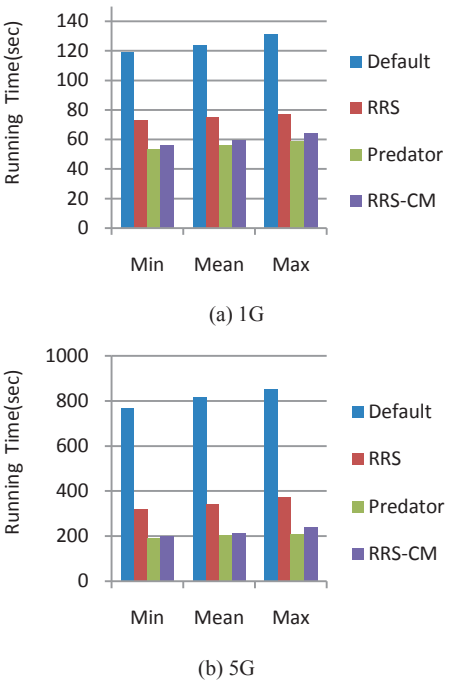
Figure 2. WordCount with different input sizes

C. *The Efficiency*

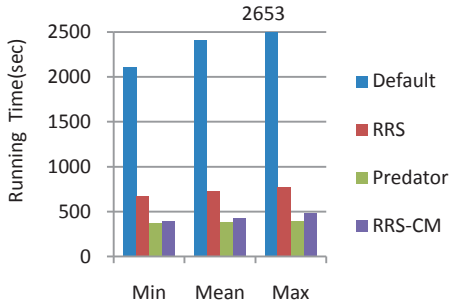
We use TeraSort MapReduce program with 1G input to evaluate the efficiency of RRS and GHC algorithms. We apply RRS and GHC algorithms respectively to search for the optimal configuration in the parameter space. Fig.4 shows the searching time and the number of searches (also the number of traversed points) for each algorithm, and each algorithm is repeated 200 times. We could easily observe the advantage of our searching algorithm GHC. The decrease in searching time and the number of searches results from the narrowed space refined by the configuration model and fairly global searching strategy of GHC. Besides, the data points of GHC in Fig.4 are more compact than the RRS. This could demonstrate the stability of GHC.

TABLE V. THE PHYSICAL CONFIGURATION OF THE CLUSTER

No.	Node	CPU	Memory
1	NameNode/JobTracker	2 cores 3.00 GHz	4GB
2	DataNode1/TaskTracker	2 cores 1.86 GHz	2GB
3	DataNode2/TaskTracker	2 cores 2.33 GHz	2GB
4	DataNode3/TaskTracker	2 cores 2.83 GHz	2GB
5	DataNode4/TaskTracker	2 cores 3.20 GHz	4GB



978-1-4673-4510-1/12/$31.00 ©2012 IEEE



(c) 10G

Figure 3. TeraSort with different input sizes

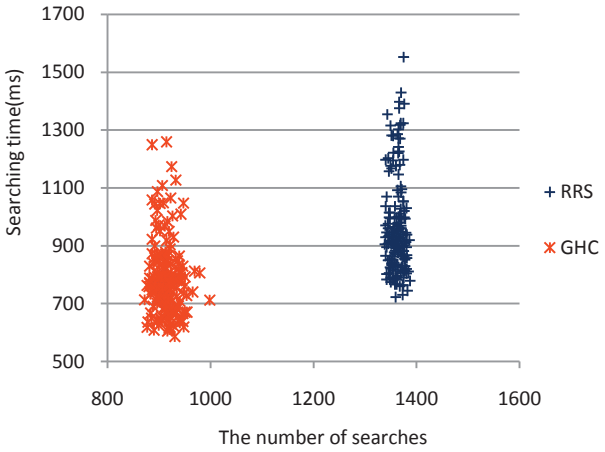


Figure 4. RRS vs GHC in searching time and the number of searches

VIII. Conclusion

By learning the practical experience of Hadoop configuration, we could optimize the Hadoop configuration with clear direction, not treating this optimization problem as a pure black optimization problem. The configuration model's preprocessing could narrow the searching space, and a Grid Hill Climbing algorithm makes the local optimum close to the global optimum by dividing the overall parameter space into equal subspaces. Experimental results demonstrate Predator's effectiveness and efficiency.

Further research could include optimization ideas from the aspect of resources allocation balancing among cluster nodes with difference physical capabilities. We are currently doing the research about MapReduce job scheduling based on the cost model established by us. Besides, designing a job description model according to different submitted jobs will make the implementation of Predator more flexible. And automatically tuning MapReduce job's configuration to optimize the job execution during the running process will also have a bright perspective.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI 04), Dec. 2004, pp. 137-150.

[2] Apache Hadoop. http://hadoop.apache.org. (2011, Nov. 20).

[3] K. Kambatla, A. Pathak, and H. Pucha, "Towards Optimizing Hadoop Provisioning in the Cloud," Proc. First Workshop on Hot Topics in Cloud Computing, June 2009.

[4] H. Herodotou et al., "Starfish: A Self-tuning System for Big Data Analytics," Proc. 5th Biennial Conference on Innovative Data Systems Research (CIDR 11), Jan. 2011, pp. 261-272.

[5] D.R. Jones, M. Schonlau and W.J. Welch, "Efficient global optimization of expensive black-box functions," Journal of Global Optimization, 13(4), 1998, pp. 455–492.

[6] "Optimizing Hadoop Deployments," Intel Corp., Oct. 2010.

[7] A Dynamic Instrumentation Tool for Java. http://kenai.com/projects/btrace. (2012, Feb. 21).

[8] B. Bonet, H. Geffner, "Planning as heuristic search," Artificial Intelligence, 2001, 129 (1–2) , pp. 5-33.

[9] B. Xi, Z. Liu, M. Raghavachari, H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," Proc. 13rd International Conference on World Wide Web (WWW 04), May 2004, pp. 287-296.

[10] A. Saboori, G. Jiang, and H. Chen, "Autotuning configurations in distributed systems for performance improvements using evolutionary strategies," Proc. 28th IEEE International Conference on Distributed Computing Systems (ICDCS '08), Dec. 2008, pp.769-776.

[11] W. Zheng, R. Bianchini, and T. D. Nguyen, "Automatic configuration of internet services," Proc. EuroSys 2007, March 2007, pp. 219–229.

[12] T. Ye et al., "Traffic management and network control using collaborative on-line simulation," Proc. International Conference on Communication (ICC 01), 2001.

[13] T.Ye, H. T. Kaur, and S. Kalyanaraman, "A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration," Proc. ACM SIGMETRICS 2003, June 2003.

[14] S. Joshi, "Apache Hadoop Performance-Tuning Methodologies and Best Practices," Proc. ACM/SPEC 3rd International Conference on Performance Engineering (ICPE 12), April 2012, pp. 241-242.

[15] Y. Li, "Analyze and optimize cloud cluster performance - Use configurable parameters to monitor and tune the performance of a cloud Hadoop cluster," IBM Corp., Mar. 2011.

[16] "Hadoop Performance Tuning," Impetus Technologies Inc, Oct.2009.

[17] H. Chen, W. Zhang, and G. Jiang, "Experience transfer for the configuration tuning in large-scale computing systems," Proc. ACM SIGMETRICS 2009, June 2009.

[18] T. Lipcon. Cloudera: 7 tips for Improving MapReduce Performance. http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce- performance. (2012, Apr. 23).

[19] H. Herodotou and S. Babu, "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs," Proc. 37th International Conference on Very Large Data Bases (VLDB 11), August 2011.

[20] X. Lin, Z. Meng, C. Xu, M. Wang, "A Practical Performance Model for Hadoop MapReduce," Proc. IEEE International Conference on Cluster Computing Workshops (ClusterW 2012), in press.

[21] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 2nd ed. Prentice Hall, 2003.

978-1-4673-4510-1/12/$31.00 ©2012 IEEE