An Intelligent Load Balancing Algorithm Towards Efficient Cloud Computing

Yang Xu, Lei Wu, Liying Guo, Zheng Chen
Agent-based Cooperative System Lab
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, SC, China, 611731

Lai Yang, Zhongzhi Shi
Key Laboratory of Intelligent Information Processing
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China, 100190

Abstract

MapReduce provided a novel computing model for complex job decomposition and sub-tasks management to support cloud computing with large distributed data sets. However, its performance is significantly influenced by the working data distributions over those data sets. In this paper, we put forward a novel model to balance data distribution to improve cloud computing performance in data-intensive applications, such as distributed data mining. By extending the classic MapReduce model with an agent-aid layer and abstracting working load requests for data blocks as tokens, the agents can reason from previously received tokens about where to send other tokens in order to balance the working tasks and improve system performance. Our key contribution lies in building an efficient token routing algorithm in spite of agents' unknowing to the global state of data distribution in cloud. We also built a prototype of our system, and the experimental results show that our approach can significantly improve the efficiency of cloud computing.

Keywords: Multi-agent, Load Balancing, MapReduce

Introduction

With the development of distributed systems, cloud computing provides a flexible and scalable approach of managing thousands of distributed heterogeneous nodes to make up integrate and high performance computing environment for users. It has been successfully applied in applications such as business, scientific research and industries. However, when the size of cloud scales up, cloud computing is required to handle massive data accessing requests, such as distributed data mining. A key challenge on those applications is that clouds have to keep the same or better performance when an outburst of data accessing request occurs, i.e., In GIS application, an area becomes hot search area when a disaster takes place, and heterogeneous nodes with different computing are unbalanced.

Introduced by Google [1], MapReduce provides a straightforward and efficient framework for processing huge data sets in cloud computing with flexible job decomposition and sub-tasks allocation. However, when applied to unbalanced cloud, its performance cannot be guaranteed. Unbalance is a key bottleneck for scalable heterogeneous cloud, while the original balancer in Hadoop is only eye on the static storage space balance, without considering the working loads attached to the data blocks. Many researches have made efforts on load balancing in grid systems, those works mainly focused on the task queues [6] or job host [7], which is not important to MapReduce framework. And some fundamental architectures no longer exist in MapReduce [9]. The load balancing in MapReduce has also been discussed by some other researchers. Some of them focused on offline adjustment with learning [2] or predict job progress [3] [4], but they are not real-time strategies; Some others tried to optimize single node to improve the whole system's efficiency [3] [5], but lack of the inter-nodes scheduling; Moreover, some others built their own MapReduce-like distributed computing frameworks, such as [1] and [4], but they are also unable to handle the dynamic computation changes in clouds. Due to the lack of real-time monitoring individual task assignments and data distribution to dynamic adjusting and rescheduling jobs to efficiently support their computations, most of previous researches cannot solve the dynamic computing load balancing problem.

Problem Description

Computation unbalanced problem in MapReduce architecture, which is popular in state of the art cloud computing can be described as follows:

There are a set of computing nodes $A = \{a_1, a_2, a_3, \dots\}$ in the cloud A. Please note, that we only specify balancing computation problem in this paper. Therefore, only working nodes are taken into account. a_i is the work node, which is the specific unit for data storage and processing. Submitted task $Task$ in MapReduce is divided into N parallel sub-tasks, then choose M ($M \leq N$) nodes are chosen to execute $reduce$ operation for the sub-tasks. MapReduce has to allocate the task on the nodes who have the computation data. When the system is unbalanced, a typical case is that a few nodes hold the most tasks to access the data allocated on those nodes.

To solve the problem, nodes with computational bottleneck are required to copy data to other nodes with bigger block of free space to share their computational tasks. In the optimal case, each node holds at most one computation task and has almost the same size of free storages.

Assuming in $Reduce$ process, sub-tasks are assigned to

the agents. Each task ts_i who requires a computation storage τ_i is assigned to an agent a_j. We can describe the assignment process as:

$$Task \rightarrow A$$

Supposed that the free space for each agent a_j is f_j, the free space updating function after task allocation is:

$$f_j = f_j - \sum_{i=1}^{m_j} ts_i.\tau_i$$

Where m_j is the number of tasks assigned to the agent a_j in $Reduce$ Process. Moreover, the computation cost of agent a_j to perform tasks it hold as is

$$c_j = (1 + k_j)^{m_j - 1}$$

k_j $\in$(0,1) is the parameter which is related the capability of the node that agent a_j represents. The system's calculation cost is scaled as C=$\sum_{j=1}^{n} a_j.c_j$. The goal of balancing working load is to reduce calculation and free storage variance and we can define the utility function as

$$R = \underset{Jactions}{argmin} \left(\sqrt[2]{\frac{1}{n}\sum_{j=1}^{n}(f_j - avg)^2} + \sum_{j=1}^{n} a_j.c_j + commCost \right) \quad (1)$$

$avg = \frac{1}{n}\sum_{j=1}^{n} f_j$ is the average working load across the system. Therefore, agents are going to search for their joint activities $Jactions$ to copy data to minimize both free space distribution variance of working load and computational cost of each nodes, inaddition to decrease the communication over network toward their cooperations, which is defined as commCost.

Agents-Aid Models

In this paper, we introduce MAS system which is suitable for complex, open and distributed problem solving into the MapReduce frame, called Agent-aid model. By representing each node, agents can communicate with each other to cooperatively manage and adjust the balance of working loads. Figure 1 illustrates the structure of agent-aid system in Hadoop, a popular cloud system. Agent-master is mainly used in centralized task processing such as task allocation, while in this paper we use completely distributed approach for adjusting load balancing and only focus the design of the MAS in agent-worker level. Agent-worker level comprises of a series of agents, which will monitor the data nodes and conduct information acquisition. The information includes node hardware performance and the actual capacity of processing task. In addition, agents will monitor nodes' working load in real time, and make the decisions on whether load balancing is necessary in the nodes that they represented. By cooperating with each other, agents should be able to jointly make decisions on how to move data blocks to minimize the utility cost function described in section 2.

Cooperative Agents for Load Balancing

In the process of load balancing, the overload nodes have to request computing resources from other nodes, and copy

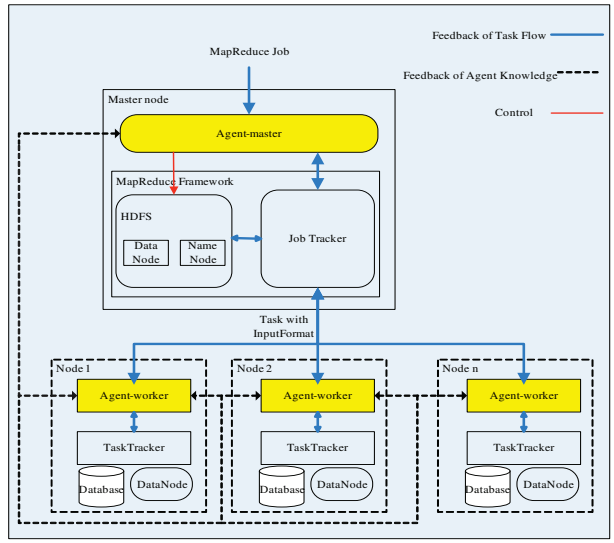

Figure 1: The architecture of agent-aid system based on Hadoop

their data to the new nodes who have more resources. Before a data migration agreement is reached, agents are required to interact with others to find who can provide those resources. In a way which is similar to our token-based coordination design [8], we abstract agents' cooperation as tokens. When an agent starts a token, it claims the request for extra working load. When an agent gets a token, it either accepts or passes it. If accept, it will provide the requested working load. Otherwise, it will transfer the request to other agent and give up responding the request any more.

Let $TOKEN$={Δ_1,Δ_2,Δ_3, ...} represents the set of all tokens. Each token contains four parts: Δ=< Resource, Path, Threshold, Lifetime>. $Resource$ identifies the block size of data for copy. $Path$ records the sequences of agents that the token visited. $Threshold$ defines the threshold that an agent is required to accept token. An agent may keep a $request$ token if its computing resource is greater than the token's $threshold$. $Threshold$ is an important part of tokens. The threshold of a token is set by its resource request sender. When the token is being passed and no agent is able to provide the resource needed, we can induce that few nodes in the system can satisfy the threshold. In this case, we make the threshold value decrease gradually until the token is kept by an agent, on the premise that the threshold value is no less than the size of the requested resource. In addition, the resource request agent can determine the initial value of threshold autonomously. The threshold value can be dynamically change and design a heuristic approach to assist resource request agent in setting the initial value of threshold. The principles are as follows:

Based on its observations when the tokens are sent back and forth or always are not kept by other agents for a long time, the agent would induct that the system was busy now, and no large blocks of computing resource existed. Thus, the agent would decrease the initial value of threshold to make

the token more acceptable by other agents. Contrarily, if the resource request agent found from its past observations that its past tokens were always easily accepted, the agent would set a higher threshold value in initiating its tokens, to improve the acceptation thresholds and make its tokens to find optimal receivers.

$Lifetime$ defines its time length allowed to exist in the network. In the transmission process, the lifetime will decrease progressively when it passes through an agent. When an agent receives a token with the end of lifetime, the agent must terminate the token's transmission, no matter whether it can provide the resources the token needed. The goal of our design is to reduce unnecessary communication cost brought by excess tokens passing. Furthermore, we may take the tradeoff between system utility and communication consumption dynamically according to lifetime.

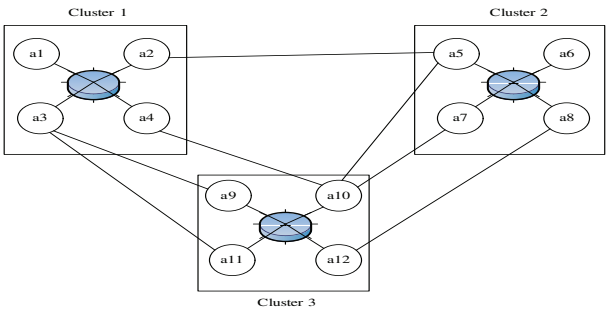

Figure 2: Agent-worker logical network relying on physical network paradigm

We build a static logic network as graph $G(A, L)$ for token movements, where A is the set of the agents and L is the set of all the edges in graph G. For $\forall a_i, a_j \in A$, $(a_i, a_j) \in L$ represents agent a_i can exchange token with agent a_j directly, and they are called neighbors. $L(a)$ represents the set of all the neighbors of agent a where $\mid L(a) \mid << \mid A \mid$. An example graph of G is shown as figure 2. The advantage of this design is that agents can gain more tokens from their neighbors to build their knowledge and infer team states so that the routing efficiency can be improved. In addition, from the manifest of small world effect, if tokens can be passed optimally, it can reach the destination with very few hops and the overall communication cost for agents' coordination is tiny.

Decision Models For Routing Tokens

In this part, we will describe our token routing algorithm. It is a process that agents attempt to minimize the overall systems cost by moving tokens around the system. Considering in a scalable cloud, with the limitation of communication bottleneck, agents cannot have the knowledge of global state of the working load distribution with flooded message sharing. With a partial observation to global state, we mode agents' reasoning process as a Partially Observable Markov Decision Process (POMDP).

The basic decision model of agent a for a token Δ can be written as a POMDP $<S, Action_a, T, \Theta_a, O, R>$. S is the state space and its specific value in time t defined as $s(t)$, $Action_a$ is the action space of a, T: $S \times A \rightarrow S$, is the transition function that describes the resulting state $s(t+1) \in S$ when executing $\chi \in Action_a$ in $s(t)$. Θ_a is the observations of agent a, and Θ_a = $<Tokens(a, \text{t}), H_a>$. $Tokens(a, \text{t})$ are all the tokens currently held by a. $H_a(t)$ records all the incoming and out-going tokens of a before t, andΘ_a include not only the tokens the agent currently holds but also all the previously incoming and out-going tokens (in H_a). The observation function is defined as O: $\Theta_a \times S \rightarrow \Omega_a$. Belief state Ω_a is a discrete probability distribution vector over the team state $s(t)$ inferred from current local state Θ_a. R: $S \rightarrow R$ defined the instantaneous reward for being in a specific state where the utility function define in section 4 decreases. This model can be applied to any agent and any token.

$Action_a$: $S \rightarrow (L(a) \cup a)$ is to move Δ to one of $L(a)$ or keep it for itself. For notation convenience, $\chi \in Action_a$ can be written as $move(\Delta, b)$ where $b \in (L(a) \cup a)$. Note that keeping a token for itself applies when the agent need the resources or able to provide resources. In general, we define a function $Acceptable(a, \Delta)$ to determine whether Δ should be kept by agent a.

$R(s(t)) > 0$ when at $s(t)$, a goal of load balancing is achieved. Team will be credited an instant rewards value when balance the data.

POMDP model provide a way that agents can reason and act optimally. However, its computation is NEXP-COMPLETE. In this paper, other than computing via POMDP model [10], we provide a heuristic model for token-based decision. P_a is the decision matrix agent a uses to decide where to move tokens. Each row $P_a[\Delta]$ in P_a represents a vector that determines the decision where to pass a token Δ to their neighbors. Specifically, each value $P_a[\Delta, b] \rightarrow [0, 1]$, $b \in L(a)$ represents a's decision that the probability of passing token Δ to a neighbor b would be the action that maximize team reward. The key to this distributed reasoning lies in how the probability model P for each agent is updated.

Token-Based Heuristic Algorithm

In this section, we provide a heuristic approach for token based load balancing. The resulting approach allows fast, efficient routing decisions, without requiring accurate knowledge of the complete global state. Initially, agents have no knowledge of their neighbors' working load distribution, therefore they have no idea where to forward the tokens. To assist their decisions, agents build their knowledge base. In this paper, we build agents' knowledge base solely from the tokens previously received. Therefore, no additional communication and observation are required. We defined a simple estimation vector for each agent a about the thresholds that its neighbors may be able to accept from agent' incoming tokens as T_a. Specifically, each element $T_a[a_i]$ represents agent a's estimation of the threshold that the neighbor a_i or the agents close to a_i is able to accept. Intuitively, the larger the free working load size is, the higher the probability for the neighbor to accept the token is. Therefore, we determine the probability of forwarding the tokens to a's

neighbors by agent a's estimations of their neighbors. P_a is normalized based on T_a.

We setup the heuristics functions for an agent a to update T_a with an incoming token Δ:

(1) If an agent gets Δ from its neighbor, the agent may infer that its neighbor or the part of network close to the neighbor may not be able to provide the working load beyond the threshold requested. Therefore, the agent is less likely to pass the other tokens with higher thresholds to that neighbor.

(2) If Δ involves with multiple neighbors, i.e., $\Delta.path =< ....,b,...,c >$ where $b, c \in L(a)$, a will adjust $T_a[b]$ and $T_a[c]$. As c is the neighbor who sends Δ, will be update according to rule (1). But for the case for neighbor b, we have to setup offset to b as the threshold when b passed the token is higher.

(3) If Δ is the token who started from a and when Δ is sent back from the accepted node, agent a will update the T_a model with the neighbor that Δ was sent to with Δ's threshold.

(4) Each agent's estimation to neighbors' threshold should be gradually increased as the time passes by some tasks are finished, new allocated working data are released.

Algorithm 1 summarizes the reasoning and token routing decision process for an agent a. The agent firstly receives all incoming tokens from its neighbors as function $getToken(sender)$ (line 2). For each incoming token Δ, a will determine whether it is acceptable with defined function $acceptable$ (line 4). If it is acceptable, a will send an $acceptNotify$ to $\Delta.path$[0] which refers to Δ's sender (line 5). If a is unable to satisfy Δ's requirement, it will countdown $\Delta.lifetime$ (line 7). If Δ reaches its lifecycle, it will be killed and no more passing over the network is allowed. (line 8-9). If a decides to pass Δ, it will add itself to $\Delta.path$ (line 11). Line 12-19 define the important procedures how T_a is updated according to the heuristic rules (1) to (3) above. After T_a has been updated, we will update the token passing probability P_a which is to normalize T_a. The large $T_a[a_i]$ is, the greater $P_a[a_i]$ is. Before Δ is passed, the value of $\Delta.threshold$ is reduced to fascinate to be accepted by the other agents except it has been minimized to the resource value it requested (line 23-24). Then a will choose to the neighbor to pass the token to according to $P_a[\Delta]$ (line 27-28).

In the following, we will explain the practical function designs on how T_a is updated. In line 14, as to the neighbor from which Δ was passed cannot satisfy $\Delta.threshold$, agent a infers that they cannot satisfy the threshold that the token asks for. Therefore, a has to update the inferred value of the threshold for the neighbors involved when a overestimates their values. In the simple case if the neighbor b is the agent who directly passed the token to a and $T_a[b] > \Delta.threshold$, we update as the mean of $T_a[b]$ and $\Delta.threshold$. For all the other neighbors that are within $\Delta.path$, we have to detect $\Delta.threshold$ when Δ was transmitted to those neighbors. Since in the transmission process, $\Delta.threshold$ has reduced by d every time Δ passes through an agent. In consequence, an offset to estimate the $\Delta.threshold$ has to be considered, which is related to the

Algorithm 1 Decision process for agent a to pass incoming tokens

```
1:  while true do
2:     Tokens(a) ← getToken(sender);
3:     for all Δ∈ Tokens(a) do
4:        if Acceptable(a, Δ) then
5:           acceptNotify(Δ.path[0]);
6:        else
7:           Δ.lifetime- -;
8:           if (Δ.lifetime≤0) then
9:              Kill(Δ);
10:          else
11:             Append(self, Δ.path);
12:             for all a_i∈(Δ.path∩L(a)) do
13:                if (T_a(a_i)>Δ.threshold) then
14:                   UpdateT(T_a[a_i],Δ);
15:                end if
16:             end for
17:             if (Δ.path[0]=a) then
18:                UpdateNotif(T_a[a_i],Δ);
19:             end if
20:             for all a_i ∈ L(a) do
21:                UpdateP(P_a[a_i],Δ);
22:             end for
23:             if ((Δ.threshold-d)>Δ.resource) then
24:                Δ.threshold -= d;
25:             end if
26:             neighbor←Choose(P_a[Δ]);
27:             Send(neighbor, Δ);
28:          end if
29:       end if
30:    end for
31:    Maintain(T_a);
32: end while
```

numbers of nodes that the token has passed through before Δ reaches a. We can calculate the value is as follows:

$$offset = d \times (\Delta.path.length - findlocation(a, \Delta.path))$$

function $findlocation(a,\Delta.path)$ returns the sequence position of a in $\Delta.path$ (starts from 1). For example, returns 4 when agent a is the fourth agent it passed. Please note that there is a special case that when $\Delta.threshold$ stopped to decreased as it reached $\Delta.Resource$ before arrived a. In this case, we only estimate their mean of offset as $o.5 \times d \times (\Delta.path.length - findlocation(a, \Delta.path))$. In summary the update function $UpdateT(T_a[a_i],\Delta)$ in line 14 can be uniformly expressed as:

$\forall\, a_i \in (\Delta.path \bigcap L(a))$, UpdateT($T_a[a_j]$,$\Delta$)=

$$\begin{cases} o.5 \times (T_a[a_j] + \Delta.threshold + offset) \\ if(\Delta.threshold > \Delta.resource) \\ o.5 \times (T_a[a_j] + \Delta.threshold + \frac{1}{2}offset) \\ if(\Delta.threshold = \Delta.resource) \end{cases}$$

In line 18, when the agent a gets the notification of Δ is accepted and supposed b is the neighbor that Δ was sent, a

will do function $UpdateNotif$ function as:

$$T_a[b] = (T_a[b] > \Delta.Threshold)\,?\,T_a[b] : \Delta.Threshold$$

a will update $T_a[b]$ if Δ reports a higher threshold. Moreover, in our algorithm, agent a's T_a cannot be always decreased as explained in heuristic rule (4). The $Maintain$ function in line 31 is defined as practical function for each maintenance period:

$$T_a = \alpha \times T_a \;\; where\ \alpha > 1.$$

If a node is overloaded, it will generate a token Δ to resort to other nodes for the data migration request. In the last part of this section, we introduce how agent a initiates Δ's threshold to balance on finding the best node and minimizing communication cost. Basically the $initThreshold$ is initiated by the formula:

$$initThreshold_a = \frac{\beta}{|L(a)|}\sum_{b \in L(a)} T_a[b]$$

Where $\beta > 1$ is an influence factor which could be dynamically adjusted and we considering two cases. If agent a's previously requests were not notified for a long time, a may predict that the system is busy and no large free data block exists. In this case, a will set β less to be easily accepted by the others. Otherwise, if a frequently gets notification of requests with big threshold, it may predict that the system may have enough computation resources. To balance to find the best node, a will adjust β higher.

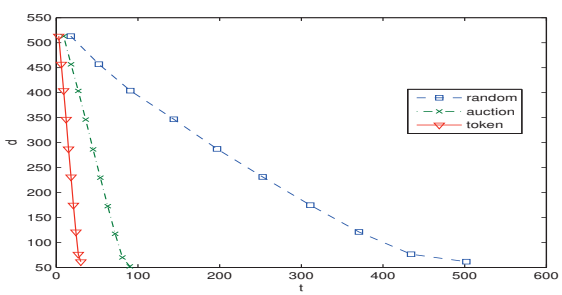

Figure 3: Token approach made the fast response on allocating working load requests and balancing computing

Experimental Results

In this section, we established a prototype system to simulate MapReduce system. This system consisted of 10000 virtual nodes. In our simulation, we abstracted the physical network of cloud computing and the logical network for routing tokens is a small world network built based on 100×100 grid. We supposed that all the nodes in the network have the same computational capacity but with different initial working loads. When the node's working load overweighs its computational capacity, it will initiate a load adjustment request. In all the three experiments, we set the randomly initial working loads in the system with the mean of 500 and each node's capacity is 1000.

It the first experiment, we investigated the performance of token-based approach by comparing with centralized approach and the random approach which passed the work load directly without using tokens. In token approach, the tokens that represented the resource requests were passed randomly to ask if the nodes could satisfy their requirements. It is notable that we did not apply our heuristic approach of algorithm 1 in choosing the capable nodes, as we set token's threshold equivalent to its resource. In centralized condition, we adopted a simple auction approach to choose the capable nodes. In this case, each overloaded node is required to ask for the computational capacities of all the other nodes. After a simple auction, the agent will forward its working load to the most capable one. While in random scheme, the data block was directly forwarded to a random neighbor regardless of its capability. We firstly chose an outlet node to send 3000 working data requests with 300 per time and ten times. At each time, the other batch will be generated after the last batch of tokens was accepted. We premised the communication bottleneck was 1000 per tick, and the communication traffic generated by token or message for each transmission was 1. The traffic for transmitting a data block was 10000. Therefore, if beyond its communication bottleneck, a node had to process the data in the next interval which would delay the system responding time. In this experiment, we investigated the variance of the unbalanced computing nodes and the elapsed time when each batch of tokens were accepted. As shown in figure 3, our token-based approach works best with the best responding time for both token acceptance and balancing the network.

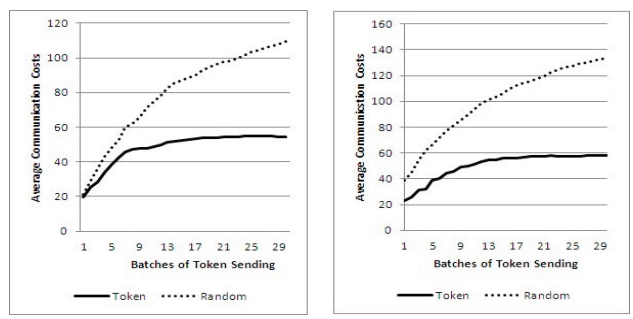

Figure 4: Tokens' average communication costs over the network in the token approach and random approach with different system sizes

In the second experiment, we compared the efficiency for tokens' finding the destination nodes using heuristic approach and random approach. In our heuristic approach, we adopted the algorithm described in algorithm 1. In the random token passing, agents did not build any knowledge but tokens were forwarded in a random way. Similar to experiment 1, the outlet node would send 30 batches of tokens with 100 tokens for each time. At each time, the other batch will be generated after the last batch of tokens were accepted. In figure 4, we investigated the average communication cost (one message per move) after each batch of tokens were ac-

cepted. In tuition, the early batches of tokens are easily accepted by agents around the outlet agents, therefore, their average communication costs are small. But with more tokens come out, they have to travel further to find capable agents. The results in random token passing matched our hypothesis. But our heuristic algorithm took the benefit from previously received tokens and could find the capable agents faster in trading with the longer distances to them. Therefore, the average communication cost almost stayed in the same level after 5 batches. Figure 4.a and 4.b showed the same conclusions with different numbers of nodes (10000 nodes in 4.a and 15000 nodes in 4.b)

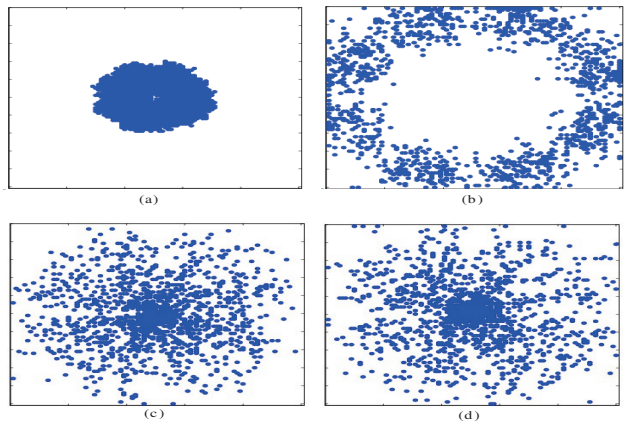

Figure 5: The token distributions over a grid network with different ways of initialling thresholds

In our third experiment, we examined the validation of dynamically adjusting tokens' initial threshold to balance the working data over the network. Unlike the previous simulations, the nodes were organized as 100×100 grid network. The outlet node was in the center of the network and sent 1200 tokens at once. Each node was set to be able to accept only one token. In this experiment, we have made three settings. (1) Tokens with very low initial threshold; (2) Tokens with very high initial threshold; (3) Tokens with random initial threshold. We hypothesized that by dynamically setting tokens' initial threshold, agents can be more efficient to balance the system. When an agent accepted a token, it was marked as a blue dote in figure 5. The results matched our expectation. In 5.a, when the tokens' initial thresholds were uniformly low, tokens were accepted very close to the outlet agent and their distribution was not balanced. In 4.b, when the tokens' initial thresholds were uniformly high, tokens were all pushed far away from the outlet agent and their distribution was not balanced as well. In 4.c and 4.d, when the tokens' initial thresholds were randomly set, tokens' distribution was evenly scattered and were balanced.

Conclusion and Future Work

In this paper, we proposed an agent-aid model by combining multi-agent system and decision-making theory toward working load balancing problem in large clouds. By abstracting agents' cooperations to locate the requests of working loads as tokens, we have built a heuristic approach to find optimal providers with agents' partial observations to global working load distributions. Our simulation results showed that agents can intelligently pass tokens to maximize the team efficiency with limited communication costs.

As the initial steps toward agent-based scalable clouding, we have a lot of challenges and questions to solve. Firstly, we believe that if our design of logical network matches the characters of physical distributions of clouds, the efficiency could be greatly improved. Finding a suitable logical network will be a nice research in the future. Secondly, importing more heuristic rules will make our model more intelligent. At last, building a real system other than abstract simulations to prove the efficiency of our approach is the most important work for us to do next.

Acknowledgement

Our work is supported by Key Projects of National Science Foundation of China (No. 61035003, 60933004, 60905042) and National Basic Research Priorities Programme (No. 2007CB311004).

Reference

1 J. Dean, S. Ghemawat. MapReduce: Simplified data processing on large clusters. In Symposium on Operating System Design and Implementation, 2004.

2 R. Bryant. Data-intensive super computing: The case for DISC. In Technical Report CMU-CS-07-128, Carnegie Mellon University, 2007.

3 M. Zaharia, A. Konwinski, A. Joseph, R. Katz, I. Stoica. Improving MapReduce performance in heterogeneous environments. In 8th USENIX Symposium on Operation Systems Design and Implementation, 2008.

4 J. Al-Jaroodi, N. Mohamed, H. Jiang, D. Swanson. Middleware infrastructure for parallel and distributed programming models in heterogeneous systems. In IEEE Transactions on Parallel and Distributed Systems, 2003.

5 M. Zaharia, D. Borthakur, J. Sarma. Job scheduling for multi-user mapreduce clusters. In Technical Report UCB/EECS-2009-55, Electrical Engineering and Computer Sciences University of California at Berkeley, 2009.

6 N. Nehra, R. Patel. Distributed parallel resource co-allocation with load balancing in grid computing. Journal of Computer Science and Network Security, 2007.

7 V. Kun-Ming, V. Yu, C. Chou, Y. Wang. Fuzzy-based dynamic load-balancing algorithm. Journal of Information, Technology and Society. 2004.

8 Y. Xu, P. Scerri, K. Sycara, M. Lewis. An integrated token-based algorithm for scalable coordination, Autonomous Agents and Multiagent Systems. 2005.

9 S. Chau, A. Wai, C. Fu. Load balancing between computing clusters. 4th Conference on Parallel and Distributed Computing Applications and Technologies, 2003.

10 A. Cassandra, M. Littman, N. Zhang. Incremental pruning: a simple, fast, exact method for partially observable Markov decision processes. In Scientific American, 2003.