Gossip-based Dynamic Load Balancing in a Self-organized Desktop Grid[*]

Sheng Di, Cho-Li Wang, Dexter H. Hu
The University of Hong Kong
Pokfulam Road, Hong Kong
{sdi, clwang, hyhu}@cs.hku.hk

Abstract

This paper presents a decentralized scheduling algorithm for dynamic load balancing in a self-organized desktop Grid environment. The proposed desktop Grid system allows volunteer nodes to join or leave freely at runtime, whereas user tasks can be launched at any node and make best use of computing resources by transparent process migration. To achieve rapid aggregation of runtime load information, we design an efficient gossip-based protocol based on an unstructured peer-to-peer dynamic network. The decentralized scheduling algorithm allows each node to determine which tasks to be relocated autonomously for achieving load balancing. Our autonomous scheduling solution can avoid the reassignment conflict problem, where different local schedulers may decide to migrate their process(es) to the same target node, by a process selection method based on game theory. The simulation results demonstrate that our solution can excel the centralized greedy scheduling algorithm and can perform as well as a meta-heuristic algorithm, while retaining small migration overhead.

1 Introduction

Desktop grid computing (a.k.a. volunteer computing) utilizes geographically distributed computers to solve complex computational problems [2, 4, 5]. A master program running on a central server is used to distribute subtasks and analyzes incoming segment-results. Such simple job pool scheme may have several drawbacks: (1) The centralized approach may suffer high synchronization overheads or long propagation delay when aggregating load information in a large-scale desktop system. (2) It may also become a bottleneck in determining an optimal task partitioning strategy and a subtask distribution plan when application logics become more complicated and types of resource become more heterogeneous. (3) It is vulnerable to the dynamically changing environment as the participating nodes may join or leave at any time without prior notice.

To address the above issues, peer-to-peer (P2P) desktop Grid has been suggested [8, 19]. In a P2P desktop Grid, each participant is not only a consumer but also a contributor, which allows others to use its own resources on a reciprocal basis. This framework makes it possible for users to find volunteer hosts in the network to run his/her tasks autonomously without relying on a centralized server. It also increases robustness in case of dynamically nodes joining and leaving. However, P2P desktop Grids also put forward two design challenges: (1) How to quickly discover the load status of volunteer hosts in the absence of a central manager and under a rather dynamic environment? (2) How to achieve robust dynamic load balancing in such opportunistic scheduling environment?

Although the task scheduling problems have been well studied in the past, existing researches [8, 10, 14, 19] seldom consider the runtime overheads in aggregating the load information. Some load balancing approaches adopted in a P2P desktop Grid collect load information through a DHT-based structured overlay network [19] or approximate flooding method over unstructured P2P network [8]. Recent studies show that gossip-based protocols [6, 12, 13] have relatively low diffusion overhead and have been proven to be resilient to rapidly renovating network, particularly when continuous node arrival and departure occurs (i.e., *churn* problem). Yet, the research on how to incorporate gossip-based load aggregation mechanism with autonomous task scheduling solution remains limited.

For the second challenge, a decentralized load balancing algorithm on an unstructured P2P Grid has been proposed in [14]. But its load migration is restricted to neighbor nodes, which may take relative long time to converge to a system-wide balanced status. Moreover, existing strategies [10, 14, 19] are mostly performed at the job submission stage. Since no runtime process migration is supported, more flexible and fine-grained load balancing policies can not be applied. A good survey on load balancing algorithms used in unstructured P2P systems can be found in [20].

[*]This research is supported by Hong Kong RGC grant HKU7176/06E and China 863 grant 2006AA01A111.

In this work, we propose a self-organized desktop Grid in which volunteer hosts could arbitrarily join/leave the network by connecting/dropping a few selected neighbors. An unstructured P2P network based on the *Newscast* model [17] is adopted to connect all the participating nodes. Based on the *Newscast* model, each node changes its network connections periodically by connecting to a subset of its peers which are randomly selected. The purpose is to maintain an approximately random topology and keep the network more scalable and robust. Moreover, it can be used to aggregate the load information more efficiently.

In the self-organized desktop Grid, there is no any central site or controller. Each user can first launch new tasks at his/her machine at any time. When any machine is overloaded, some of its tasks could transparently migrate to other nodes via process migration [9]. To achieve dynamic load balancing, each node will independently perform a rank-based autonomous scheduling algorithm to determine which process(es) should be migrated to reduce its workload or whether it should accept external migrated processes to share the workload of other busy nodes. The solution has to meet the two contracting goals: (1) best load balancing effect (e.g. standard deviation or makespan) (2) minimum migration cost. In our design, each node periodically exchanges load information with its neighbors based on an epidemic gossip protocol [6]. Besides, we deploy an aggregation gossip protocol [7, 18] for computing and propagating a few critical global statistics (e.g. average load) which are required in our autonomous scheduling algorithm. Given a tolerable statistics error, the proposed aggregation gossip protocol is able to aggregate the required global statistics within a small number of load exchange iterations over the unstructured peer-to-peer network.

Based on the load information gossiped on each participating node, each autonomous scheduler makes use of fuzzy-based ranking functions to evaluate the effectiveness of load balancing among a shortlisted process migration plans with low migration cost. Our solution is different from traditional fuzzy-based dynamic load balancing algorithms [22], which may lead to poor scalability due to the use of the central fuzzy controller. Moreover, we try to avoid the *reassignment conflict* problem, in which different local schedulers may decide to migrate their process(es) to the same target nodes, by designing a selection rule based on game theory [21]. We enforce the final process migration decision be fixed from a set of derived candidate solutions.

Simulation results show that our dynamic load balancing solution can perform much better than the centralized greedy algorithm and very close to optimal result estimated by the centralized Markov Chain Meta-heuristic (MCM) algorithm. The total computation time of our solution is just 10% of the MCM. Moreover, the whole average migration cost can also be bounded to 1.12 seconds per node for all testing cases in our benchmark.

The rest of this paper is organized as follows. In Section 2, we define and formulate the dynamic load balancing problem in a self-organized decentralized desktop Grid and give an overview of the proposed algorithm. We discuss our rank-based scheduling algorithm in Section 3. The algorithm complexity and simulation results are analyzed in Section 4. Finally, we conclude our work in Section 5.

2 Gossip-based Load Balancing Algorithm

In this section, we first define the objective function of the load balancing problem, then give an overview of our proposed gossip-based dynamic load balancing algorithm, abbreviated as GB-DLB.

2.1 Problem Definition

In the basic model, we assume the total number of nodes is n and the total number of processes running in the system is m. For the purpose of illustration, both n and m remain unchanged during the execution. Let g_i denote a grid node i, where 1⩽i⩽n. Assume at node g_i, there are m_i processes time-sharing its computing resource. Let p_{ij} represent the jth process on g_i, where 1⩽i⩽n and 1⩽j⩽m_i. Process p_{ij} will produce a workload of d_{ij}, where d_{ij} could be p_{ij}'s CPU percentage cost, memory usage, or their combination. Let c_i be the capacity of g_i and it could be viewed as memory capacity or the computation capacity of g_i measured by FLOPS according to practical cases.

We define dc_i as the *load level* [8, 15] of a specific node i, where dc_i=$(\sum\limits_{j=1}^{m_i} d_{ij})/c_i$, $\sum\limits_{j=1}^{m_i} d_{ij} \leqslant c_i$. We use the root-mean-square deviation of the load level σ, where σ=$\sqrt{\frac{1}{n}\sum\limits_{i=1}^{n}(dc_i - \overline{dc})^2}$, to evaluate the load balancing effect of different process migration alternatives. We also consider the migration overhead: If the process p_{ik} is migrated from g_i to g_j, the cost of migration mig_cost_{ikj} can be calculated as $\frac{mig\_procsize(p_{ik})}{bw_{ij}}$, where bw_{ij} is the network bandwidth between g_i and g_j.

Given an initial assignment with a total of m processes running on n desktop nodes, our goal is to achieve a minimum σ with the lowest migration cost $\overline{mig\_cost}$.

2.2 Overview of GB-DLB Algorithm

For simplicity, asynchronous cycle-driven communication mode is used in our design. The whole execution period needs to be split into multiple *epochs* and each epoch contains h *gossip cycles*. Each node exchanges load states among its neighbors every gossip cycle. The goal of this "epoch with gossip cycles" design is to adapt to aggregation gossip protocol [7, 18] which makes every node obtain accurate statistics within h gossip cycles in dynamic network. These statistics are mainly used to guide the succeeding load balancing decision. To ensure the convergence speed of the target aggregated values can be obtained, nodes join/leave are only treated after the h gossip cycles. The aggregation gossip protocol is performed over an unstructured P2P network constructed based on the *Newscast* model [17], where every node is connected to a fixed number c of neighbors. In each gossip cycle, every node merges its neighbor set with that of one randomly selected neighbor. An updated neighbor set is created by randomly selecting another c nodes from the merged set. The aggregation gossip messages will be sent/received along these new neighbors, thus a dynamic network will be constructed accordingly, improving the accuracy of aggregated statistics.

Let $d_{p\min}$ denote the workload of the smallest process among the m processes in the whole system. In our algorithm, we distinguish overloaded and underloaded nodes according to Formula (1), where $\overline{dc}$ denotes the average load level of the n participating nodes.

$$g_i\ is \begin{cases} underloaded \cdots\cdots if\ dc_i < \overline{dc} - \frac{d_{p\min}}{2c_i} \\ overloaded \cdots\cdots if\ dc_i > \overline{dc} + \frac{d_{p\min}}{2c_i} \\ balanced \cdots\cdots otherwise \end{cases} \tag{1}$$

We consider a node is in a *load balanced* state if its current load is very close to $\overline{dc}$ and can not be further improved by migrating any process into it or out from it. Accordingly, the scheduler launched on that node should be suspended once the node's load level is in the range of $[\overline{dc} - d_{p\min}/(2c_i), \overline{dc} + d_{p\min}/(2c_i)]$ until a new *epoch*.

Assume all nodes know the system scale n initially. Our GB-DLB Algorithm will perform aggregation gossip protocol and load scheduling in every *epoch* iteratively.

In each gossip cycle, every node asynchronously computes the specified aggregation functions based on the values received from its randomly selected c neighbors and then sends the results (attached in an aggregation message *ag-m*) back to the neighbors (Line 5$\sim$6). Thus, global statistics, such as the average load level ($\overline{dc}$), system scale (n) and the minimum process workload ($d_{p\min}$) can be aggregated at each node during this course. To avoid unnecessary traffic overhead, each node sends its own state-update message (*su-m*) to its neighbors (Line 9) only if there is an observable change over threshold dc_t to its load level (Line 7).

The autonomous scheduler (Line 22, which will be described in Section 3) on each node will be executed after all the gossip cycles in every epoch, performing process migration to balance workload. It will be ignored if the node's load level has already turned into *load balanced* state based on Formula (1) (Line 20).

Algorithm 1 Skeleton of Gossip-based DLB algorithm

*/*This algorithm is executed on each node.*/*

```
 1: while (TRUE) do
 2:     /*Each while-loop body is an epoch with $h$ gossip cycles*/
 3:     for each gossip cycle in the epoch cycle do
 4:         Randomly reselect $c$ neighbors based on Newscast model.
 5:         Receive gossip messages from $c$ neighbors: including aggregation msg(ag-m) and state-update msg(su-m).
 6:         Compute the new aggregation values ($\overline{dc}$ and $d_{p\min}$) and send them to neighbors back.
 7:         if load level change of the node $> dc_t$ then
 8:             Compute $hops\_distance_u$ or $hops\_distance_o$ based on Formula (1).
 9:             Encapsulate su-m and send it to its neighbors.
10:         end if
11:         for each su-m received from neighbors do
12:             Store su-m in uNList or oNList in terms of whether su-m is underload information or overload information.
13:             su-m's $hops\_count$ ++.
14:             if $hops\_count <$ su-m's $hops\_distance$ then
15:                 Forward su-m to its neighbors.
16:             end if
17:         end for
18:         Sleep until next gossip cycle.
19:     end for
20:     if Current node is overloaded according to Formula (1) then
21:         Search its uNList for underloaded nodes.
22:         Perform Rank-based Autonomous Scheduler.
23:     end if
24:     Sleep until next epoch cycle.
25: end while
```

In our design, each node keeps one *uNList* and one *oNList* to record which nodes nearby are underloaded and overloaded respectively. Nodes in uNList are the candidates for hosting the execution of migrated jobs, while nodes in oNList are used to evaluate the overloaded status of current node compared to others. The two lists can be enriched over a few gossip cycles, and each record refers to a node's state, represented as a 3-tuple: $\{node\_ID$, *freshness*, *state*$\}$. The freshness value is computed as $hops\_count \cdot \gamma + \lambda$, where γ is an estimated transmission latency and λ is the elapsed time since the node status was received. The smaller *freshness* value, the fresher the record is. If the status information is outdated (i.e., *freshness* is larger than a given threshold value), it will be discarded.

To prevent the information from being excessively spread, the number of hops for a message to propagate is computed in Line 8. For those underloaded nodes, their node status message can only pass over $hops\_distance_u$ hops, where $hops\_distance_u$=$\omega \cdot (1 - dc_i)$; for overloaded nodes, $hops\_distance_o$=$\omega \cdot dc_i$ (ω refers to the upper-bounded number of hops for each message). Each state-update message (*su-m*) is a 4-tuple: $\{hops\_count$, $hops\_distance$, g_i's ID, $dc_i\}$, where $hops\_count$ is the number of the hops it has walked. For each node, whenever it receives a state-update message from its neighbor, it will store it to uNList or oNList locally and increment

$hops\_count$. If $hops\_count<hops\_distance$, the node will forward it to some of its neighbors.

To determine h for each epoch, a necessary and sufficient condition of getting exponential convergence speed in aggregation gossip protocol has been proven by [16] to be $\frac{E(\sigma_{i+1}^2)}{E(\sigma_i^2)}=\rho$, or $\frac{E(\sigma_h^2)}{E(\sigma_0^2)}=\rho^h=\varepsilon \Rightarrow h=\log_\rho \varepsilon$, where σ_i^2 and σ_{i+1}^2 respectively refer to the variance at the ith and (i+1)th gossip cycles, $\rho(\leq 1)$ is a constant only dependent on the network characteristic and ε is an expected decline ratio to the initial variance of load level, σ_0^2. In our design, we organize a dynamic *Newscast* model [11] (Line 4) in which $\rho=\frac{1}{2\sqrt{e}} \approx 0.303$ in theory [16]. However, $\frac{1}{2\sqrt{e}}$ may not be perfectly met in practical cases, especially when the average node's degree is not big enough. Based on our experiments (Section 4.1), we may restrict $\rho<0.4$ and limit c to O($\log_{10} n$). We also estimated the required ε as 2.78×10^{-4}. Hence, h's upperbound for the chance of not reaching the required ε is $\log_{0.4}\varepsilon$=8.94 $\approx$9. That is, in reality, there is supposed to be at least 10 gossip cycles in each epoch.

3 Rank-based Autonomous Scheduler

In this section, we discuss an autonomous scheduler, which makes process migration decisions based on a set of rank functions, each of which is composed of several evaluation functions. We introduce evaluation functions first.

3.1 Evalution Functions

To achieve load balancing, we need to decide which process(es) should be migrated and where they should be executed. The optimal solution is hard to find as it is a well-known NP-complete problem. In reality, there is a cost for task migration, including time to capture the execution state, network delay in moving the captured context, and time for restarting the task at the destination node. A good load balancing algorithm should be efficient in computing the best task remappings that could achieve balanced workload and also consider the overall migration overheads.

As the desktop Grid is basically heterogeneous, moving a process from one node to another may impose different loadlevel changes to them. For example, if d_{ik} refers to the workload of the selected process p_{ik}, it will add $\frac{d_{ik}}{c_j}$ load to g_j, while g_i's load is reduced by $\frac{d_{ik}}{c_i}$. Hence, each node's capacity must be considered seriously. Plus, we also need to take into account how to lower migration cost, so finding a content heuristic solution becomes very complex. To suitably combine/compare the different factors above, a leveling function $lev(x)$ is defined: $lev(x) = \lceil x \cdot L \rceil / L$, given $x \in [0,1]$ and $\lceil\rceil$ is the *ceiling function*. L is the number of $levels$ which can be tuned flexibly. The leveling function tries to map the value of x to L discrete levels.

We define three evaluation functions (f_1,f_2 and f_3) to help finding the most appropriate migration solution for each overloaded node. In order to restrict scheduling time, the migration decision will be performed in two steps at each overloaded node: (1) select one or more target underloaded nodes based on f_2 and f_3; (2) choose appropriate processes to migrate to some of these selected nodes based on f_1 and f_3. The three evaluation functions are defined as follows.

- $f_1(g_i,g_j,p_{ik}) = lev(\frac{MIN\{|\varphi-d_{i1}|,|\varphi-d_{i2}|,\ldots,|\varphi-d_{im_i}|\}}{|\varphi-d_{ik}|})$
  $where\ \varphi = \frac{c_j^2(d_i-\overline{dc}\cdot c_i)-c_i^2(d_j-\overline{dc}\cdot c_j)}{c_i^2+c_j^2}$
- $f_2(g_i,g_j,p_{ik}) = 1 - |lev(rank_o) - lev(rank_u)|$ where
  $rank_o = \frac{dc_i-\frac{d_{ik}}{c_i}}{MAX(dc_o-\frac{d_{ik}}{c_o})}, rank_u = \frac{MIN(dc_u+\frac{d_{ik}}{c_u})}{dc_j+\frac{d_{ik}}{c_j}}$
- $f_3(g_i,g_j,p_{ik}) = lev(\frac{min\_mig\_cost}{mig\_cost\{p_{ik}\rightarrow g_j\}})$ where
  $min\_mig\_\cos t = \frac{min\_mig\_procsize(p_{iq}\ on\ g_i)}{bw_{i\max}}$
  $and\ mig\_cost\{p_{ik} \rightarrow g_j\} = \frac{mig\_datasize(p_{ik})}{bw_{ij}}$

In order to assess different processes p_{ik} (1$\leq$k$\leq m_i$) on an overloaded node g_i, $f_1(g_i,g_j,p_{ik})$ (where g_j is a specific target underloaded node) is deduced as follows: Compared with σ's value before and after p_{ik}'s migration from g_i to g_j, we just need to make sure $\sqrt{\frac{(\frac{d_i}{c_i}-\overline{dc})^2+(\frac{d_j}{c_j}-\overline{dc})^2+S}{n}} \geq \sqrt{\frac{(\frac{d_i}{c_i}-\frac{x}{c_i}-\overline{dc})^2+(\frac{d_j}{c_j}+\frac{x}{c_j}-\overline{dc})^2+S}{n}}$, and try to get F(x)=$((\frac{d_i}{c_i}-\overline{dc})-\frac{x}{c_i})^2+((\frac{d_j}{c_j}-\overline{dc})+\frac{x}{c_j})^2$ as small as possible according to the standard deviation of load level (σ) formula, where x refers to any process's workload d_{ij} on node g_i, thus, $x\leq 2\frac{c_j^2(d_i-\overline{dc}\cdot c_i)-c_i^2(d_j-\overline{dc}\cdot c_j)}{c_i^2+c_j^2}$ and $\frac{d(F(x))}{d(x)}$=0 $\Rightarrow$ φ=$\frac{c_j^2(d_i-\overline{dc}\cdot c_i)-c_i^2(d_j-\overline{dc}\cdot c_j)}{c_i^2+c_j^2}$ is x-axis value of the minimum point. Hence, σ would get smaller after p_{ik}'s migration if and only if d_{ik} is in (0, 2φ); process workload d_{ik} is supposed to be as close to φ as possible to get smaller σ for better load balancing. Therefore, we only select s ($\leq m_i$) processes running on g_i with load in the range of (0, 2φ) to be acted on $f_1(g_i,g_j,p_{ik})$ (i.e. $d_{ik} \in (0, 2\varphi)$).

In order to evaluate different underloaded nodes (g_j, j$\neq$i) based on their total workloads and capacities, $f_2(g_i,g_j,p_{ik})$, where g_i refers to the overloaded node and p_{ik} is assumed to be a specific process on it, is devised based on non-cooperative game theory [21], compared to the self-interest strategy on other related works [8, 19]. Basically, if every participating node always selects target nodes as idle as possible independently, reassignment conflicts may happen with high probability especially in the low latency network environment. That is, even more serious imbalance situation may occur in some areas until some of nodes give up their original decisions in each epoch. Therefore, each overloaded node should select target underloaded nodes based on their demands on resources. The overloaded node with assessed overload level $rank_o$ prefers the target nodes which have approximate underload level $rank_u$, where $rank_o$= ($dc_i - \frac{d_{ik}}{c_i}$)/*MAX*($dc_o - \frac{d_{ik}}{c_o}$) and $rank_u$=*MIN*($dc_u + \frac{d_{ik}}{c_u}$)/($dc_j + \frac{d_{ik}}{c_j}$). This design can effectively avoid reassignment conflicts. *MAX*($dc_o - \frac{d_{ik}}{c_o}$) is the maximum value of $dc_i - \frac{d_{ik}}{c_i}$ among all overloaded nodes; likewise, *MIN*($dc_u + \frac{d_{ik}}{c_u}$) is the minimum value of $dc_j + \frac{d_{ik}}{c_j}$.

Based on the migration cost definition ($mig\_cost_{ikj} = \frac{mig\_procsize(p_{ik})}{bw_{ij}}$), migration overhead is related to both the process workload and the bandwidth between the two relative nodes, g_i and g_j. We accordingly design $f_3(g_i, g_j, p_{ik})$ as follows. First of all, we find the minimum migration cost ($\frac{min\_mig\_procsize(p_{iq}\ on\ g_i)}{bw_{i\max}}$) from the m_i processes on g_i and l underloaded nodes stored in *uNList*. Then, each process (p_{ik}) to be migrated from g_i to g_j is evaluated by comparing its cost with the minimum migration cost. bw_{ij} is the bandwidth between g_i and g_j; $bw_{i\max}$ refers to the maximum bandwidth between g_i and any other nodes in its *uNList*. $mig\_procsize(p_{ik})$ is p_{ik}'s $image\ size$. Intuitively, f_3 indicates that the processes with lower migration overhead will be more favored.

Basically, our goal is to devise an approach which schedules the process migration by combining the above three factors. Otherwise, the time complexity of the selection procedure will be $O(m_i{\cdot}l{\cdot}\tau)$ which may consume much computation time, where m_i refers to the number of processes on g_i, l is the size of uNlist, and τ is the time cost on calculating rank functions. Thus, as mentioned previously, we split the whole scheduling procedure into two steps: selecting target underloaded nodes ($rank_1(\omega_1$,g_i,g_j,w_{ij})) shown as Formula (2) before choosing process(es) on g_i based on the selected nodes ($rank_2(\omega_2$,g_i,g_j,w_{ij})) shown as Formula (3). In these two steps (formulas), ω_1 ($\in$[0,1]) and ω_2 ($\in$[0,1]) are two weight coefficients. Since the first step does not consider process workload f_2 demands, we choose *AVG*(g_i) as an estimated value for the process in $rank_1(\omega$,g_i,g_j,w_{ij}). Finally, the two rank functions are both in normality format: the values are bounded in (0,1), and higher value means higher rank.

$$\begin{aligned} & rank_1(\omega_1, g_i, g_j, w_{ij}) = \\ & \omega_1 \cdot f_2(g_i, g_j, AVG(g_i)) + (1-\omega_1) \cdot lev(\frac{bw_{ij}}{bw_{i\max}}) \\ & where\ AVG(g_i) = \frac{\sum_{k=1}^{m_i} d_{ik}}{m_i} \end{aligned} \quad (2)$$

$$\begin{aligned} & rank_2(\omega_2, g_i, g_j, p_{ik}) \\ & = \omega_2 \cdot f_1(g_i, g_j, p_{ik}) + (1-\omega_2) \cdot f_3(g_i, g_j, p_{ik}) \end{aligned} \quad (3)$$

3.2 Rank-based Autonomous Scheduling Algorithm

Algorithm 2 Rank-based Autonomous Scheduler (RAS)

*/*This algorithm is executed on each node g_i.*/*

Input: dc_i,$d_{p\min}$,$\overline{cd_i}$,*uNList*,*oNList*,where $d_{p\min}$,$\overline{dc_i}$ are aggregated values.

Output: Select and migrate one of g_i's processes outward for load balancing.

```
1: while TRUE do
2:    Clear candidate_list.
3:    if $g_i$ is detected to be overloaded according to Formula (1) then
4:       uN_region_search($g_j$, candidate_list, $g_i$'s uNList).
5:       if candidate_list.length $< v_0$ then
6:          uN_sampling_search($g_i$ ,candidate_list).
7:       end if
8:       if candidate_list is empty then
9:          continue. // wait for next cycle
10:      else
11:         $process\_mig(g_i, candidate\_list)$.
12:      end if
13:   end if
14: end while
```

The pseudo-code of rank-based autonomous scheduling algorithm (*RAS*) is shown in Algorithm 2. Line 2 is to clear *candidate_list*, which is used to keep the migration candidates with target process and underloaded nodes selected later. Line 4 is to search possible underloaded nodes in nearby region (but not just among neighbor-connected nodes). v_0 at Line 5 stands for the *capacity* of *candidate_list*. If *uN_region_search()* cannot fill up *candidate_list*, then randomly sample some possibly far away *underloaded nodes* and search their nearby region records (i.e. *uNList*) for continuity. Finally, the final decisions are made in the *process_mig()* function (Line 11). In fact, the condition at Line 3 will be a little bit over-strict when most of nodes become load balanced over time. Hence, for further load balancing, dc_i>$\overline{dc}$+$\frac{d_{p\min}}{2c_i}$ can be replaced by dc_i>dc_j then, but the migration cost may increase accordingly.

In *uN_region_search()*, we first traverse all underloaded records stored in g_i's uNList and select those with highest $rank_1$ (Formula (2)) as candidates. And then, check if g_j would become overloaded as we add the smallest process to it. If not, check each target underloaded node candidate selected and find the processes to migrate from g_i in terms of $rank_2$. As long as the *candidate_list* is filled up, the searching will be terminated. Otherwise, the *uN_sampling_search()* will be activated to randomly select a far-away node for further searching.

The *process_mig()* function is used to make the final decision (which process of the current node g_i will be migrated to which node). To further avoid the *reassignment conflict* problem mentioned before, we compute a composite *rank level* for each candidate decision based on func-

Table 1. Initial Setting

Metric	Value or Scale	
	Group 1	Group 2
Node ID (i)	i = 1∼ [n/2]	i = [n/2]+1∼n
Capacity (c_i)	400∼500	100∼200
No. of processes (m_i)	7∼10	0∼2
Process workload (d_{ij})	5∼60	5∼20
Process image size	1M∼10M	1M∼10M
Bandwidth (bw_{ij})	0.1M/b∼10M/b	

Table 2. Experiment Parameters

Metric	Value or Scale
warmup for aggregation	10 cycles
warmup for state gossiping	5 cycles
f_0 (rank criterion)	$0.6 \sim 0.8$
Capacity of *candidate_list*	3
Sampling times	10
Neighbor set size c	$5\log_{10}(n)$
The no. of gossip cycles in each epoch	10
w_i (i=1,2,3)	w_1=w_2=w_3=0.5
the maximum number of hops (ω)	15

tion f_{ikj}: $f_{ikj}(\omega_3,g_i,g_j,p_{ij})=\omega_3 \cdot f_2(g_i,g_j,w_{ij})+(1-\omega_3)\cdot f_3(g_i,g_j,w_{ij})$, and then randomly select one candidate with a probability proportional to the rank level f_{ikj}, where $\omega_3(\in[0,1])$ is a coefficient to tune the weights of f_2 and f_3. Since we have selected the most proper process for each candidate node in previous steps, function f_1 related to process workload is not supposed to be considered in f_{ikj}.

4 Performance Evaluation

We use *PeerSim* [3] to do the simulation. Through *Brite topology generator* [1] , we construct an emulated physical network with n computers randomly connected with various bandwidths by *Waxman* model. For simplicity, transmission latency (γ) between any two nodes is assumed to be uniform and the same as gossip cycle period (i.e. 50ms). After constructing the network, numerous tasks are randomly initialized. The initial setting and parameters are listed in Table 1 and Table 2. Considering the worst initial assignment (i.e. maximizing the standard deviation of load level (δ)), we equally split all nodes to two groups and set them either overloaded or underloaded as initial state. All the random values (e.g. capacity) conform to uniform distribution in their ranges. Without loss of generality, any process image size is assumed to be in the range from 1M to 10M and bandwidth to be in [0.1M/b,10M/b].

In our benchmark,we first evaluate the environment used for simulation, and then, compare our RAS algorithm with the other existing solutions, including *neighbor-migration (NM)*, *greedy heuristic (GH)* and *Markov Chain Meta-heuristic*. *NM* is a P2P algorithm which only migrates processes between connected neighbors. The other two which can know global information during execution period try to get the best load balancing level, but without considering the migration cost issue.

4.1 Analysis of Experiment Environment

Through the experiments based on [3] among $10^2 \sim 10^6$ participating nodes, we evaluated that $\rho<0.4$ in a *Newscast*-based network when $c = 5\log_{10}(n)$. Moreover, through our experiments based on various random distributions and logical parameters (excluding unpractically extreme cases), we found that without any load balancing, the standard deviation of load level is always in the range from 0.2 to 0.6 and its optimal value estimated by MCM load balancing algorithm is in 0.01∼0.04. Hence, without loss of generality, the expected ratio ε to the initial value for the standard deviation of load level could be set as $\frac{E(\sigma_h^2)}{E(\sigma_0^2)}=\frac{0.01^2}{0.6^2}=2.78\times10^{-4}$.

4.2 Evaluation of RAS Algorithm

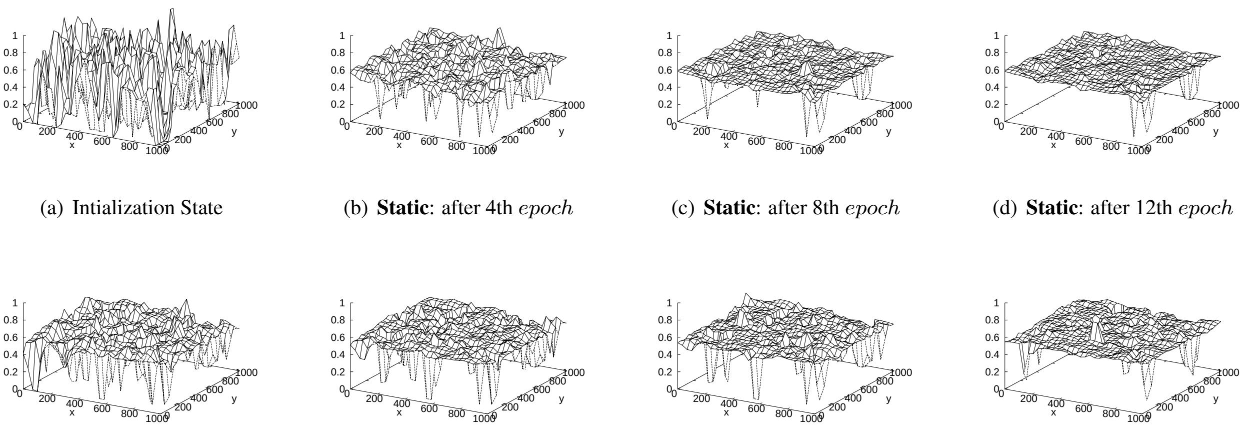
Figure 1 presents a set of snapshots with *RAS* algorithm tested in a desktop Grid with 500 nodes. Figure 1(a) shows the initial chaotic load distribution. Other sub-figures show the whole load distribution may get remarkably balanced after 8 epochs in static network or after 16 epochs in dynamic network, in which 2% of nodes are replaced in every epoch.

Other than traditional methods, each node autonomously makes decisions in every epoch in our algorithm. As shown in the Figure 2 which corresponds to Figure 1, load balancing can be reached over a few epochs and its convergence is in exponential speed no matter in static network, as well as in dynamic network, where 2%, 5%, 10% of nodes are replaced in every epoch respectively.

The standard deviation of load level (δ) is the most important metric to evaluate the load balancing effect. The benchmark result is shown in Figure 3. Because each node cannot get the global but a very shortsighted vision in *NM*, this approach always gets the worst result, whose standard deviation is much higher than others'. Since *MCM* is a *meta-heuristic* which can avoid local optimum to a certain extent, it always gains the best load balancing level, and *GH*'s result is close to it. *RAS* is always better than *GH*, though a little inferior to *MCM*.

For completeness, we also compare the final converged *makespan* computed by the four approaches, as shown in Figure 4. No doubt, *MCM*'s result is always the lowest, while *RAS* is between *GH* and *MCM*, even better than *MCM* occasionally.

We present the average migration cost in Figure 5. *GH* and *MCM* produce largest overhead and *NM* costs about half amount of them. In *RAS*, the average migration cost observed at each node is less than 1.12 seconds for all testing cases. Even in a large-scale system (n=2000), the migration



(a) Intialization State (b) **Static**: after 4th *epoch* (c) **Static**: after 8th *epoch* (d) **Static**: after 12th *epoch*

(e) **Dynamic**: after 4th *epoch* (f) **Dynamic**: after 8th *epoch* (g) **Dynamic**: after 12th *epoch* (h) **Dynamic**: after 16th *epoch*

Figure 1. Snapshots of Load Balancing Effect (*RAS*: a desktop Grid with 500 nodes)

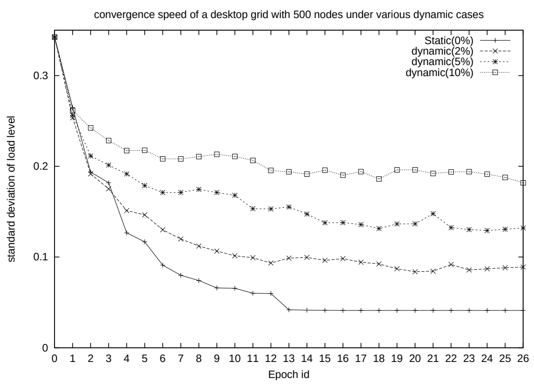


Figure 2. Convergence Speed of RAS

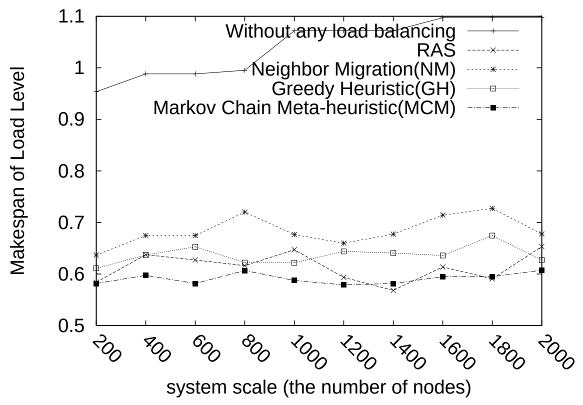


Figure 4. Makespan of Load Level

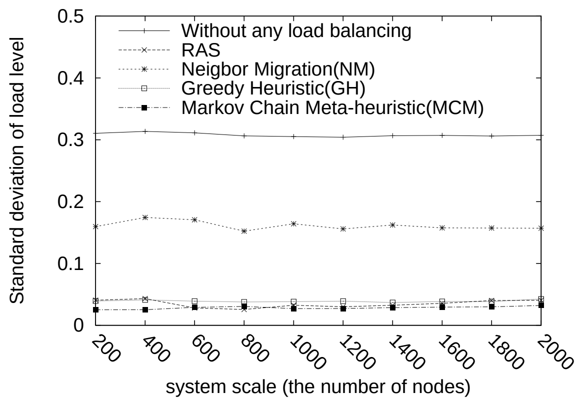


Figure 3. Standard Deviation of Load Level

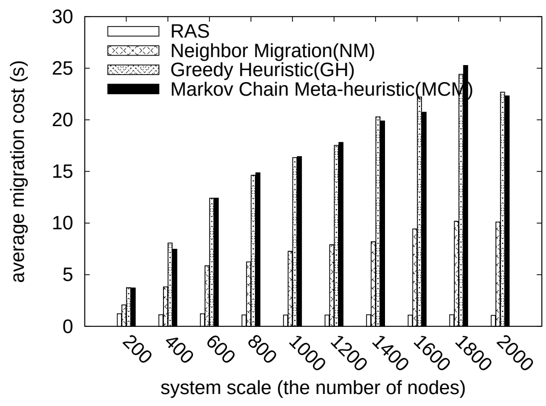


Figure 5. Average Migration Cost

cost of *RAS* is only 10% and 5% of those measured by *NM* and *MCM* respectively. This is mainly because we filter out the target nodes with low bandwidth in evaluation function f_3 so that the nodes with the bandwidth $\geq$8M/b are selected with higher probability. We also observe that the number of processes migrated is in the range from 2.3 to 2.5 per node for all testing cases.

We also compare the computation times of load balancing algorithms. Our algorithm is much better than *MCM* though a little worse than the others. *MCM* consumes the longest time, from 1 to 12 seconds, to iteratively search the relatively optimal solution among the solutions computed by *GH*. *NM* consumes the least time, from 0.003 to 0.036 second, as each node just considers their neighbors according to simple greedy selection and these operations can be concurrently executed by each node. Our algorithm is also run by each node in parallel, costing from 0.016 to 0.10 second in each epoch, about 10% of the time cost by *MCM*.

5 Conclusion and Future Work

In this paper, we design a gossip-based approach to solve dynamic load balancing in self-organized decentralized desktop Grid environment. Our focus is on how to get the best load rebalancing effect with minimized process migration overhead in such environment. We overcame a set of challenges, such as restricting transmission overhead and avoiding reassignment conflict phenomenon. Our approach is suitable for dynamic and large-scale environment due to gossip protocol. With our rank-based autonomous scheduling algorithm, each node can autonomously make effective decisions on process migration at runtime based on our mathematical analysis. By intensive simulation, we prove our algorithm always outperforms the traditional global greedy algorithms. Our load balancing effect (makespan and standard deviation of load level) is very close to the approximately optimal solution (estimated by Markov Chain Meta-heuristic) and the average migration overhead can be well limited. As a future work, we shall improve our strategy for complex demands, such as with more data dependencies.

References

[1] Brite topology generator: http://cs-pub.bu.edu/brite/.

[2] Folding@home project: http://folding.stanford.edu/.

[3] Peersim simulator: http://peersim.sourceforge.net.

[4] Seti@home project: http://setiathome.berkeley.edu/.

[5] World community grid:http://www.worldcommunitygrid.org.

[6] A. Allavena, A. Demers, and J. E. Hopcroft. Correctness of a gossip based membership protocol. In *24th Annual ACM Symposium on Principles of Distributed Computing*, pages 292–301, New York, NY, USA, 2005.

[7] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Gossip algorithms: design, analysis and applications. In *24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1653–1664, 2005.

[8] J. Bustos-Jimenez, D. Caromel, A. di Costanzo, M. Leyton, and J. M. Piquer. Balancing active objects on a peer to peer infrastructure. In *25th International Conference of the Chilean Computer Science Society*, pages 1–7, 2005.

[9] L. Chen, T. C. Ma, C. L. Wang, F. C. M. Lau, and S. P. Li. G-JavaMPI: a grid middleware for transparent mpi task migration. In *Engineering the Grid: Status and Perspective, Nova Science*. American Scientific Publishers, 2006.

[10] M. Franceschelli, A. Giua, and C. Seatzu. Load balancing on networks with gossip-based distributed algorithms. In *46th IEEE Conference on Decision and Control*, pages 500–505, 2007.

[11] A. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, February 2003.

[12] Z. Haas, J. Halpern, and L. Li. Gossip-based ad hoc routing. In *21st Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1707–1716, 2002.

[13] K. A. Hawick and H. A. James. Modeling a gossip protocol for resource discovery in distributed systems. In *Parallel and Distributed Processing Techniques and Applications*, June 2001.

[14] J. Hu and R. Klefstad. Decentralized load balancing on unstructured peer-2-peer computing grids. In *5th IEEE International Symposium on Network Computing and Applications*, pages 247–250, 2006.

[15] E. Jeannot and F. Vernier. A practical approach of diffusion load balancing algorithms. In *Euro-Par 2006 Parallel Processing*, pages 211–221, 2006.

[16] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.

[17] M. Jelasity and M. van Steen. Large-scale newscast computing on the internet. Technical Report IR-503, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, October 2002.

[18] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science*, pages 482–491, 2003.

[19] J. S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Using content-addressable networks for load balancing in desktop grids. In *16th International Symposium on High Performance Distributed Computing*, pages 189–198, New York, NY, USA, 2007.

[20] Z. J. Li and M. H. Liao. Modeling load balancing in heterogeneous unstructured p2p systems. *Journal of Computer Science*, pages 323–331, 2005.

[21] M. J. Osborne. *An Introduction to Game Theory*. Oxford University Press, USA, August 2003.

[22] C. Park and J. Kuhl. A fuzzy-based distributed load balancing algorithm for large distributed systems. In *2nd International Symposium on Autonomous Decentralized Systems*, pages 266–273, April 1995.