A Competence-based Scheduling Method for Web Computing

Chungnan Lee, MinHong Horng, and Chuanwen Chiang

Department of Computer Science and Engineering
National Sun Yat -Sen Universit
Kaohsiung, Taiwan

E-mail: cnlee@mail.nsysu.edu.tw

J. K. Wu
National Kaohsiung Institute of Marine Technolog
Kaohsiung, Taiwan

Abstract

In this paper, we proposed a heuristic-scheduling algorithm, called Competence Based Method (CBM), to dynamically distribute workload for task allocation problem in a Web/Java-based computing environment. CBM is designed to meet four properties of Web computing: heterogeneity, scalability, centralization, and non -dedication host. CBM can be applied to some independent/dependent links scheduling problems. Compared with the competing algorithms, the experimental results show that the proposed CBM performs faster in parallel time and is closed to the optimal one.

Keyword: *Task Allocation, Scheduling, Distributed computing, WWW, Java*

1. Introduction

As the Web connects rich computational resources in an unprecedented scale for easy access, it has become an important infrastructure for parallel/distributed computing (PDC). Simultaneously, the Java [1] has gained an important role for the support of Web-based parallel/distributed computing. As one of many types of distributed computing, the Web/Java-based computing involves various issues, such as dynamic execution, heterogeneity and portability, security, load balancing, task scheduling, and fault tolerance, etc. These traditional issues, however, should be concerned further with a span opinion due to the properties of the Web/Java-based run-time environment [3,5,13]. The following sketches illustrate the environment-dependent properties of the Web/Java-based computing.

Property 1. The machines in the Web/Java-based computing environment are heterogeneous. Even the same task, if executed on different machines, requires different execution time.

Property 2. The number of machines in the environment is not fixed. A remote machine may participate or depart via the Web.

Property 3. The scheme of the Web/Java-based computing is based on a centralized control. This is limited by the security consult of Java-enabled Web browser.

Property 4. Some no -deterministic natures exist in such an environment. As we know, a network system is usually non -dedicated, thus the remote site could be a multi-users, time sharing or multitasking system and its computing power could be adjusted according to local circumstances. Additionally, network transmission rate is also no n-deterministic because of the communication contention.

Based on the properties, the methodology of the Web/Java-based computing has attracted many researchers. For example, Charlotte [2] raised many key issues about Java computing, like dynamic execution, heterogeneity and portability, and security. With respect to load balancing and fault tolerance, Charlotte also proposed an eager scheduling and two-phase idempotent execution strategy. DAMPP [3] is another Web-based system. It claims that supercomputing can be attained by leaguing a swarm of client machines all over the world together. Javelin [4], a Java-based infrastructure for global computing, permits users to upload applet for extending the serviceability of Web-based Java computing. Although uploa ding applet raises the security problem, it stands for indispens ability of dynamic loading and execution. The same issue is addressed as soft-installation in IceT [5]. This project incorporates sophisticated techniques of Java and has the ability to dynamically merge virtual machines belonging to multiple users. In addition, Keren and Barak [6] focus on parallel computing in scalable computing cluster using Java agents and use asynchronous invocations to propose a scheme for adaptive placement of multiple a gents (load index and migration decision).

While these studies provide fundamental paradigms for Web/Java-based computing, little attention has been paid to task allocation problem. The task allocation problem involves job partition and scheduling. The job partition problem deals with how to dig out parallelism and determine grain size (trade-off between granularity and overhead). Although partitioning is imperative in PDC, we don't pursue this issue in this paper. Unlike partitioning, the scheduling problem assumes that a set of target machines are available and there exist a set of tasks to be served by these target machines according to certain scheduling policy. The scheduling policy is not only to determine which tasks are to be allocated to which machines, but also to determine the execution order of each task such that the tasks can be completed in the shortest time.

There are many approaches, such as queuing theory, graph theoretic approaches, mathematical programming, and state-space search, that can be employed for the scheduling problem solution. The existed scheduling algorithms are not all suitable in a Web/Java-based run-time environment. For instance, the list scheduling, a classical approach, is to make ordered lists by priority assignment an d then repeatedly executes the designated steps until schedule is finished. Many algorithms are based on list scheduling heuristic so that they simply think about which task should be scheduled first except where a task should be scheduled on [7-9]. Such scheduling methods may not be good enough because of the mentioned Property 1. Furthermore, they often finish overall tasks scheduling first and then to distribute tasks [10-12]. This approach may suffer some losses due to Properties 2 and 4 (e.g. machines' slowdown or crash). These all potentially make the scheduling worse and system inefficiency.

In this paper, we propose a heuristic scheduling algorithm: Competence Based Method (CBM) to distribute tasks on a heterogeneous, scalable, non -dedicated, and centralized Web/Java-based run-time environment. The remainder of this paper is organized as follows. In Section 2, we define a mathematical model for Web/Java-based computing by covering our basic assumptions and definitions. In Section 3, Web/Java-based distributed scheme for PDC is presented. Section 4 introduces a heuristic algorithm for scheduling called Competence Based Method (CBM). The results of performance evaluation that related to different scheduling schemes are presented in Section 5. In the last section, we give conclusions.

2. Framework and Definitions

In this section, we briefly discuss the conceptual model of the Web/Java-based computing used to study our proposed scheduling problem first. As illustrated in Figure 1, the conceptual model is divided into three participating entities coordinator, clients, and hosts. A coordinator is a server that offers some service to clients and coordinates suppl ies and demand s for computing resources. It can connect any kind and any number of idle machines a cross the Internet. A client demands and receives service from a coordinator. A host H joins with others in a distributed computing environment and devotes its computing resources to a computing environment via the Web browser . It is a non-dedicated machine and could be a PC or a workstation. We denote the number of hosts b $\underline{h}$, which is assumed to be finite. We also use the notation $\underline{H} = \{H_1, H_2, \ldots H_h\}$ to represent a set of hosts joining in a distributed computing environment.

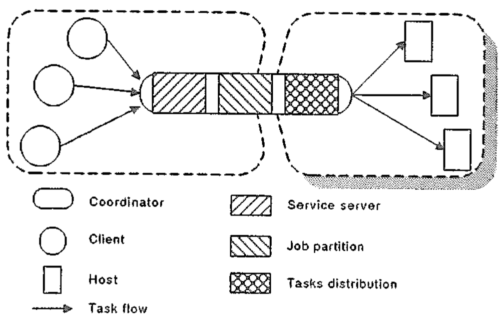

Figure 1. The conceptual model of the Web/Java-based distributed computing used for the proposed scheduling problem.

The coordinator receives requests from clients, called job, and distributes sub-jobs, called tasks, to hosts. We denote tasks as $\underline{T} = \{T_1, T_2, \ldots T_t\}$ and the number of tasks by $\underline{t}$, that is assumed to be finite too. As shown in Figure 1, the coordinator should partition the job into several tasks before distributing the jobs to the hosts . We assume that a job has been presented in the form of t ask graph.

We consider two kinds of task graphs in this paper. One is dependent task graph, the other is independent task graph. A dependent task graph is called a directed acyclic task graph (DAG) generally. A DAG is defined as a three tuple, $G = (\underline{T}, \rightarrow, WL)$.

- $\rightarrow$ is a partial order, or precedence relation, on a set of tasks $\underline{T}$ which specifies that if $T_1 \rightarrow T_2$ then T_1 is a parent of T_2, in other words T_2 is a child of T_1, and T_1 must be finished before T_2 can be scheduled to start. A task is said to be a "mature" task if it has no parent or if all of its parents have been finished.
- *WL* is a workload function, which is associated with the set of tasks $\underline{T}$ and is written as

$$WL(\text{job}) = WL(\underline{T}) = \sum_{j=1}^{t} WL(T_j) \cdot$$

Furthermore, the height of a DAG is the maximum level of all tasks. The level of a task T_j is the length of the longest path from T_j to an "exit task" and an exit task is a task that has no child. In addition, the recursive definition of tasks' level is

$$L(T_x) = \underset{T_j \in Children(T_x)}{Max} \{L(T_j)\} + 1 ,$$

where $Children(T_j)$ is the set of all children of T_x and the level of an exit task is equal to one. Taking Figure 2 as an example, the nodes I, J, K, L, M, N, and O are exit tasks and the level of node B is equal to four.

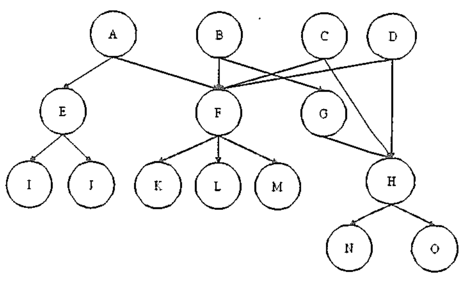

Figure 2. A sample of directed acyclic task graph.

The definition of DAG is slightly different from other literature. The communication cost parameter usually is considered in the literature, but can't be determined before schedule (Property 4). Because the execution time can't be determined before s chedule (Properties 2 & 4), we used workload instead. Before we explain how to obtain the workload of a task latter, we still have to define the other type of task graph: independent task graph. Independent task graph is somehow similar to Single Program M ultiple Data (SPMD) model or can be viewed as a special case of DAG, when its "height" is equal to one. Therefore it is called "independent" task graph.

3. Distributed Scheme of the Web/Java -based Computing

The goal of scheduling is to minimize the overall finishing time of the computation on either dependent task graph or independent one (e.g. the finishing time of the last task completed). The execution time of the entire job is called the "parallel time" or schedule length. A scheduling algorithm takes t wo inputs: H and T for different distribution schemes. Some other parameters may be derived from H and T. Figure 3 illustrates the conceptual model of the distribution scheme in this paper. The main theme to be discussed here is how to predict task execution time. Though it is based upon traditional distributed/parallel system techniques and paradigms; it brings up several novel properties as stated in Section 1 due to the characteristics of Java programming technologies like: spawning processes on remote hosts, sandbox of security, Java byte -code crossing platform, and merging/splitting hosts dynamically We predict tasks' execution time by estimating tasks' workload, evaluating hosts' computing power, and predicting communication delay between a coordinator and a host. For tractability w define some symbols first.

- *CP* is an execution speed function. *CP*(H_i) represents the computing power of the i-th host. In this paper, w use H_i to stand for an arbitrary host.
- *CD* is a communication cost function. *CD*(H_i) represents the communication dela coefficient between the coordinator and the i-th host
- *RET* is a time function. The real execution time of a task T_j on a host H_i is defined as: *RET*(H_i,T_j). In this paper we use T_j to stand for an arbitrary task.
- *PET* is a prediction function of possible execution time. We use *PET*(H_i,T_j) to denote the predicted execution time that H_i finishes T_j.

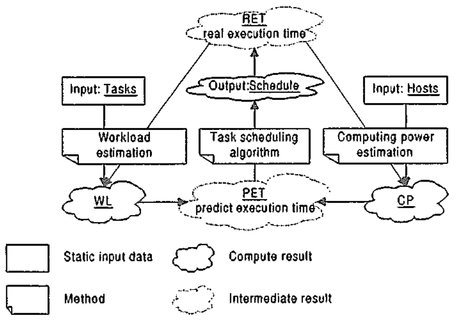

Figure 3. The conceptual distribution scheme.

The basic idea in Figure 3 consists of three stages: (1) information collection phase, (2) scheduling and allocating phase, and (3) feedback phase. In the information collection phase, the information about the related "agent" and "data" of tasks, and the hosts in the Java computing pool are used to generate a set of parameters including *WL*, *CP*, and *CD*. We present several procedures for the evaluation of computing power and the estimation of task load. The term "agent" is often used in Java computing to indicate a Java class byte -code. A Java distributed computing environment is treading toward an object world, which composes of distributed agents. Those agents are blobs of intelligence that can live anywhere on the Internet and have some special functions. Hosts can download agents via the Internet and then invoke some functions of them to finish tasks that the coordinator assigned. This technology wa named soft-installation. The end-users may extend the functions of environment by adding a distributed agent across networks. Hence, a task is associated with an agent and correspondent computation data We denote a task as T_j = {agent, data} , where T_j, j = 1...t, is atomic. In other words, a task T_j is indivisible, so that we don't partition a task while scheduling. Additionally, once a task is assigned to a host, ot her hosts cannot preempt the same task (no preemption). Both duplication of task in separate hosts and computation migration are also not taken into consideration.

In order to predict the time consumption that machines spend on a task, we have to estimate task load and computing power. Many researches about task load estimation can be found in [13]. Generally, it is difficult to estimate the execution time before running a task due to conditional statements and loop constructs whose iteration counts and conditional values are run-time variables. There are two ways about load estimation in PDC: instruction counting and profiling. Here we use profiling, because it's hard for an agent or a compiled program to count instructions.

For tractabilit , we assume that a virtual task T_v is a fixed task with known workload and is taken as a baseline of workload estimation; a virtual machine H_v is a fixed and dedicated machine with known computing power and is taken as a baseline of computing power evaluation. H_v also can be a server for task profiling. Then the computing power of an arbitrary host H_i can be calculated as follows

$$CP(H_i) = \frac{RET(H_v, T_v)}{RET(H_i, T_v)}, \qquad (1)$$

where *RET*(H_i,T_v) means the real execution time of T_v on the host H_i. When a host joins into a distributed co mputin pool, the coordinator assigns T_v to measure its initial computing power using Eq. (1). And when a distributed agent "A" is added to the distributed computing environment, we can generate a task T_j = {"A", "Data"}, where "Data" depends upon the profiling methods.

Regarding the profiling method, we classify it into three kinds: random sample, incremental sample, and characterizing matching. But no matter how exact we do, there are still man factors to affect the accuracy of predicted execution time. For example, the attenuation of computing power, the deviation of estimated workload, and network contention are all uncontrollable. The results they generated are always incorrect, however as long as we know which task is heavy, which is light, it will help us in scheduling. Thus and so, an estimation method is alternative within an acceptable error rate. In [14], Yan pointed out that the prediction errors of 10 -20% are acceptable since the predictions need not to be quantitativel accurate. In what follows, three kinds of reasonable approaches are described.

The first approach, random sample, is to generate sample data randomly. Feeding an agent with the sample data, we can estimate the computation density of an agent "A" according to the execution time. Suppose T_j = {"A", sample data}, the workload of T_j can be calculated by the following equation

$$WL(\mathrm{T_j}) = \frac{RET(\mathrm{H_v}, \mathrm{T_j})}{RET(\mathrm{H_v}, \mathrm{T_v})}. \qquad (2)$$

Eq. (2) means that a task, T_j, with unknown workload can be estimate by executing it on H_v. After we obtain every task's relative workload and every host's computing power, we can define the predicted execution time.

$$PET(\mathrm{H_i}, \mathrm{T_j}) = \frac{WL(\mathrm{T_j})}{CP(\mathrm{H_i})} * RET(\mathrm{H_v}, \mathrm{T_v}) \cdot \qquad (3)$$

In this approach, we omit the influence of input data and so there is no difference among each pair {agent, data}. For example, let a task $T_{profiling}$ = {agent "A", sample data}, a task T_{runtim} = {agent "A", runtime data}, $RET(H_v,T_{profiling})$ = 50 sec., $RET(H_i,T_v)$ = 5 sec. and $RET(H_v,T_v)$ = 10 sec., then $PET(T_{runtime}, H_i)$ equals twenty five seconds, because $WL(T_{profiling})$ equals $WL(T_{runtime})$ in a random sample approach. The prediction error of random sample approach is

$$Error \le \frac{(Max - Min)}{Min}, \qquad (4)$$

where *Max* and *Min* are the maximum and minimum of $RET(H_i,T_j)$, respectively.

The profiling server in the second approach - incremental sample, will execute every task, T_j = {agent "A", runtime data}, unless the profiling of agent "A" converges in the range being defined. Every time the profiling server executes T_j, we compute the computation density of an agent by averaging the execution t ime. Before sampling in real cases, an initial value of an agent's computation density is indispensable. The initial value of computation density of an agent can be generated by the random sample approach. Once we get a practical sample point, the logical sample point generated by the random sample approach can be kicked out. For example, let T_l = {agent "A", runtime data}, $WL(T_l)$ = 2, averaged times = 1, RE (H_v,T_v) = 50 sec. and $RET(H_v,T_l)$ = 150 sec., then $WL(T_l)$ equals (2+150/50)/(1+1) = 2.5. This approach is better than the first one, but has larger overhead. In general, every logical possible input data occurred evenl . Ultimately the computation density of an agent equals $\int_a^b f(x) \big/ (b-a)$, where b is the upper bound of logical input data and a is the lower bound of logical input data, after being executed enough times. It is trivial about prediction error, which is restricted to

$$Error \le \frac{(Max - Min)}{(2 * Min)}. \qquad (5)$$

To avoid increasing the complexity of the analysis, sometimes we tackle a big problem using a simple way. When dealing with computationally intensive problems, it becomes data insensitive problems, hence two approaches mentioned above may be efficient enough. However, for better accuracy, the third approach is used , but more complicated. Before proceeding to discuss the characterizing matching method, we briefly classify agents' parameters into three categories in accordance with their influences on execution time (1) Quantity: Since the matrix multiplication complexity is $O(n^3)$, the matrix size has a direct ratio on execution time. (2) Value: Threshold is on behalf of it. Some applications need a threshold to determine when it should be ended. (3) QV: In the field of image processing, it is common for a parameter to influence execution efficiency on both quantity and value. Using a smoothing operation as an example, ho much time it takes depends upon the image size and the degree of smoothing operati on the image. The rudiment of the third approach is to quantify the types of parameters, so we have to define some rules for quarrying agents' characteristics that are necessary for estimating task load. The typical and simplest means is equal-space. For every parameter, its data-type is known, but both its data-type and data-size may be limited within some range. So it divides parameter's value-space equally if it is classified as Value and divides its size-space equally if it is classified as Quantity. For example, let the range of an integral parameter be equal to or greater than 0 and less th an 100 and distance equal 10. Hence, the sampling points are 0, 10, 20, 30 ··· 90.

In the scheduling and allocating phase, the coordinator allocates a task to an appropriate host at pertinent time based on those parameters generated by previous phase. When some hosts' *CP* or *CD* changes, even crashes, it should adjust itself to reflect the real status. As to the scheduling method, we will discuss it in Section 4.

Eventually, in the feedback phase, because $CP(H_i)$ and $CD(H_i)$ have dynamic behavior, and load esti mation cannot be precise, the predicted execution time is not always equal to real execution time. They must be tuned to real change dynamically for each interval. We re-calculate the computing power b

$$CP(\mathrm{H_i}) = \frac{PET(\mathrm{H_i}, \mathrm{T_j})}{RET(\mathrm{H_i}, \mathrm{T_j})} * CP(\mathrm{H_i}) \cdot \qquad (6)$$

Using the informatio n of physical execution, we estimate computing power, communication delay, and task load repeatedly. As a result, we can use the discrepancy between previous state and current state to make a better schedule.

4. The Competence-based Scheduling Algorithm

The fundamental concepts of CBM are that we must schedule the task to have the most impact on the parallel time first and incorporate the properties mentioned in Section 1. Hence, there are two strategies to be involved: task selection and host selection. We trace the scheduling step by step owing to some dynamic factors of Property 4 mentioned in Section 1. In other words, we don't decide the next task to assign until the hosts finished the tasks. At the same time, we also re-calculate relative scheduling parameters. In the following, we introduce the CBM and explain how it works on independent task graph and then adapt it to dependent task graph. Ultimately, the adapted CBM can tackle independent/dependent task graph.

As stated in Section 2, in the scheduling problem of independent task graph, we assume that a job has been decomposed into a number of tasks, $\underline{T} = \{T_1, T_2, \ldots T_t\}$, that can be executed in any order in a number of hosts, $\underline{H} = \{H_1,$

H_2, ⋯ H_h}. The goal is to minimize the overall finishing time of t he computation. We now introduce a set of definitions to be used in the description and analysis of our proposed scheduling method

- $\underline{T}_{exec}$: Let T_{exec} denote that a task has been scheduled but not finished, $\underline{T}_{exec}$ is a set of T_{exec}.
- $\underline{T}_{unmature}$: Let $T_{unmature}$ denote that a task is not mature, $\underline{T}_{unmature}$ is a set of $T_{unmature}$.
- $\underline{T}_{wait}$: Let T_{wait} denote that a task is already mature and yet to be scheduled, $\underline{T}_{wait}$ is a set of T_{wait}.
- $\underline{T}_{occupied}$: Let $T_{occupied}$ denote that a task can't be assigned to one host because of a nother host with more competence, $\underline{T}_{occupied}$ is a set of $T_{occupied}$.
- $WL(H_i,T_{exec})$=
$$\frac{\left|WaitTime(H_i,T_{exec}) + PET(H_i,T_{exec}) - SpentTime(\underline{T})\right|}{RET(H_v,T_v)} * CP(H_i),$$
where $WaitTime(H_i,T_{exec})$ is waiting time from the moment that a coordinator starts scheduling $\underline{T}$ to the moment that H_i starts executing T_{exec}. And $SpentTime(\underline{T})$ is the time from start-up time of scheduling to current time. If T_{exec} is not yet executed, $WaitTime(H_i,T_{exec})$ equals $SpentTime(\underline{T})$.
- $Balancing(H_i)$=
$$\frac{CP(H_i)}{\sum_{n=1}^{h} CP(H_n)} * \left(\sum_{T_j \in \underline{T}_{wait}} WL(T_j) + \sum_{T_j \in \underline{T}_{exec}} WL(agnHost(T_j), T_j) \right),$$
where $agnHost(T_j)$ is the host that is assigned by T_j.
- $Competence(H_i,T_{wait})$=
$$\frac{\sum_{\forall T_{exec} \text{ assigned to } H_i} WL(H_i, T_{exec}) + WL(T_{wait}) + \sum_{\forall T_{occupied} \text{ occupied by } H_i} WL(T_{occupied})}{CP(H_i)}.$$

Before discussing the scheduling method to minimize parallel time required to execute a task graph, we state two facts for a task to be scheduled to a host.

Fact 1.

Necessary - condition for scheduling a task to a host: Assume that a task T_{wait} is considered to be scheduled to a host H_i. If it satisfies any of the following conditions

(a) $WL(T_{wait}) + \sum_{\forall T_{exec} \text{ assigned to } H_i} WL(H_i, T_{exec}) \le Balancing(H_i)$

(b) $Competence(H_i,T_{wait})$=
$$Min(\forall \text{host} \in \underline{H}, Competence(\text{host}, T_{wait}))$$
then T_{wait} can be assigned to H_i.

Fact 2.

Necessary condition for scheduling a tas k to a host: Assume that a host H_i consider to select a task for execution. For each task in $\underline{T}_{wait}$, if there is a task T_{wait} that satisfied Fact 1, then H_i could select T_{wait} for execution.

The general idea behind this heuristic is to use two parameters for the tasks' allocation to the hosts. The *Competence* parameter is used to evaluate the competence of one host H_i for one task T_j. It takes the current load of H_i, the workload of T_j, and the summation workload of occupied tasks into account totally. If th e host H_i has the best competence to do the task T_j, it implies that $RET(H_i,T_j)$ is minimum compared with other hosts. This is a standard greedy strategy to find minimum but with high time complexity. As a result, we need another parameter to fast judge whether the assignment is valid. The reason to use the *Balancing* parameter is to make the tasks scheduling faster. We take it as a kind of loose competence. If the total load of host H_i is below the *Balancing* itself, it means that H_i couldn't be the encumbrance for minimizing the parallel time. Consequently, we can assign the task T_j to the host H_i in constant time if the sum of current load of H_i and the workload of T_j is equal to or less than the *Balancing* of H_i. The competence-based method algorithm for ind ependent task graph is as follows

The CBM Algorithm

Input: A set of tasks, $\underline{T}$ = {T_1 T_2 ... T_t}, and a group of hosts, $\underline{H}$ = {H_1 H_2 ...H_t}.

Output: A task schedule of $\underline{T}$ on $\underline{H}$.

1. Initially, every T_j is a member of $\underline{T}_{wait}$.
2. Use workload and computing power as the pri ority of tasks and hosts respectively, and ties are broken arbitrarily.
3. If $\underline{h}$ is greater or equal to the number of $\underline{T}_{wait}$, go to Step 6, else use *Fact 2* to select a task for each host to execute.
4. For each host that has been allocated a task in Step 3, if those hosts amount is greater or equal to the number of $\underline{T}_{wait}$, go to Step 6, else use *Fact 2* to select one more task to execute on those hosts.
5. If a host finishes a task, select another task to execute using *Fact 2*. Repeat Step 5 until $\underline{T}_{wait}$ is empty.
6. For each task in $\underline{T}_{wait}$, schedule T_{wait} using *Fact 1(b)*. If the host has been allocated more than two tasks, defer schedule time.

We present CBM that finds a solution when $\underline{H}$ is non-dedicated and using communication and computation hiding technology. *Balancing(.)* and *Competence(.)* functions are computed repeatedly for making schedule decision. It responses possible variation of hosts' status. Whenever a host joins or departs, or whenever the computing power gains or loses, it works. If $\underline{t} > \underline{h}$, CBM seeks to reduce co mputation complexity by the threshold function, *Balancing(.)*. This is because we think that the dominate function to minimize the overall finishing time of the computation is in tuning and balancing the terminal stage of scheduling.

Now we extend the previous analysis of CBM to DAG. Suppose that a job has been decomposed into a number of tasks, $\underline{T}$ = {T_1, T_2, ... T_t}, and with some precedence-constrained among tasks and there are a number of hosts, $\underline{H}$ = {H_1, H_2, ⋯ H_h}, in the distributed computing environment . The problem of scheduling a weighted directed acyclic graph to a set of hosts is called dependent tasks scheduling problem. The objective is to assign the tasks of the DAG to the hosts such that the schedule length is minimized without violating the precedence constraints. We can modify the CBM to adapt it to DAG. Because a computation of independent tasks can be viewed as a DAG with one level, the CBM can be applied to independent/dependent tasks scheduling problem. Of course, the adapted CBM is as efficacious as the original one. We redefine *Balancing(.)* and *Competence(.)* functions before discuss the adapted CBM.

- The recursive definition of tasks' accumulated workload is
$acWL(T_j)$ =
$$WL(T_j) + Max\{\forall \text{Task} \in Children(T_j), acWL(\text{Task})\},$$
where $acWL(T_{exit\_task})$ is equal to $WL(T_{exit\_task})$.

- *Balancing*(H_i) =
$$\frac{CP(H_i)}{\sum_{n=1}^{h} CP(H_n)} * \left(\sum_{T_j \in \underline{T}_{wait}} acWL(T_j) + \sum_{T_j \in \underline{T}_{exec}} WL(agnHost(T_j), T_j) \right),$$
where *agnHost*(T_j) is the host, which is assigned by T_j.
- *Competence*(H_i, T_{wait}) =
$$\frac{\sum_{\forall T_{exec} \text{ assigned to } H_i} WL(H_i, T_{exec}) + acWL(T_{wait}) + \sum_{\forall T_{occupied} \text{ occupied by } H_i} WL(T_{occupied})}{CP(H_i)}$$

Based on the definitions above, the adapted CBM algorithm for dependent tasks scheduling problem is presented as follows

The adapted Competence Based Method
Input: Tasks of DAG, $\underline{T} = \{T_1\ T_2\ \ldots\ T_t\}$, and a group of hosts, $\underline{H} = \{H_1\ H_2\ \ldots H_t\}$.
Output: A task schedule of $\underline{T}$ on $\underline{H}$.
1. Initially, every T_j is a member of $\underline{T}_{unmature}$.
2. Use accumulated workload and computing power as the priority of tasks and hosts respectively, and ties are broken arbitrarily.
3. Add each mature task into $\underline{T}_{wait}$.
4. Schedule those tasks in T_{wait} by CBM.
5. If a host finishes a task, go to Step 3 until $\underline{T}_{wait}$ and $\underline{T}_{unmature}$ are empty.

5. Performance Evaluation

So far, we have presented a distribution paradigm in Java computing with scalable, heterogeneous, web -based distributed environment and analyzed tasks scheduling problem. In this section, we conducted several simulations to evaluate the performance and efficiency of CBM. Firstly, we examine whether the CB M is efficiency enough for the scheduling problem of independent task graph. This part involves another heuristic algorithms: Heavy First Method (see appendix [I]) , which uses $WL(T_j)$ and $CP(H_i)$ as a task's and a host's priorities, respectively. Moreover, t o evaluate the performance of the heuristic algorithms under independent task graph case, we use a brute -force approach, which searches all possible schedules exhaustively to obtain the optimal result for the purpose of comparison. We compare the results obtained from the heuristic algorithms with the optimal one. The simulation patterns used to evaluate the algorithms are listed in Figure 4.

```
int[] CP = {15,30,45};                      // random seed for CP
int[] WL = {15,30,45};                      // random seed for WL
int[] HostNum = {2,3,4,5,6,7};              // No. of hosts
int[] TaskNum= {15,10,8,7,6,6};             // No. of tasks
ran r = new ran();                          // a user defined class

for (Int i = 0;i < CP.length;i++)
      for (Int j = 0;j < WL.length;j++)
            for (Int k = 0;k < HostNum.lengthk++)
            {
                  r.genHost(HostNum[k],CP[i],(5*CP[i])/7);      // randomize CP
                  r.genTask(TaskNum[k],WL[j],WL[j]/11);         // randomize WL
                  BruteForce();                                 // obtain the optimal schedule
                  CBM();                                        // Competence Based Method
                  HFM();                                        // Heavy First Method
            }
```

Figure 4. The simulation patterns for independent task graph (part 1).

In Figure 4, there are fifty-four groups including a wild variety of *CP*, *WL*, $\underline{h}$, and $\underline{t}$ matchmaking. For each new test, a new set of test data randomly is generated. The *genHost* function randomizes a set of computing power according to appointed quantity and random seed. To avoid generating illogically data, we restrict the ratio of maximu *CP* over that of minimum *CP* to be less than six among randomizing *CP*. We believe that six is reasonable because the speed of the latest PC, Pentium III, is about six times better than th at of the primitive Pentium machine.

Similarly, the ratio of maximu *WL* minus minimu *WL* is also constrained to be less or equal 20% of minimum *WL* in the function *genTask*. This is because we want to observe the influence that both heavier and lighter tas ks left on hosts. Assuming that *WL* and *CP* of test patterns are divided into minimum, medium, and maximum, respectively, there are nine kinds of matchmaking like {minimum *CP*, minimum *WL*}, {minimum *CP*, medium *WL*}, and so on.

After the evaluation of the finishing time over all test cases under various conditions of all scheduling algorithms, we define function *Efficiency(.)* for an algorithm as follows

$$Efficiency(S) = \frac{\text{Parallel time under Competence Based Method (CBM)}}{\text{Parallel time under the scheduling algorithm S}} \quad . (7)$$

The above equation indicates the efficiency of algorithm "S" is defined as the ratio of parallel time ratio of the CBM to that of the algorith "S". As shown in Figure 5 and Figure 6, the efficiency for the brute force is about one for all test data, in other words, the performance of the CBM algorithm is close to the performan ce of the brute force. On the contrary, the efficiency for the HFM algorithm is less than one for more than one half of test data. Hence, it is obvious that the proposed CBM algorithm performs better than HFM for independent tasks scheduling problems.

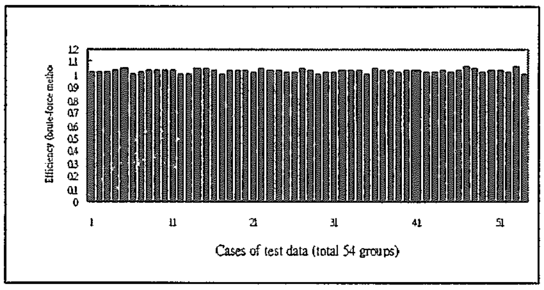

Figure 5. The efficiency of the brute-force method in the first simulation.

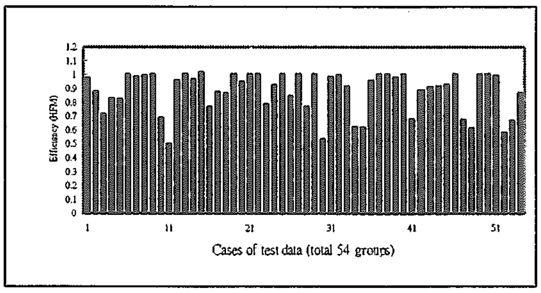

Figure 6. The efficiency of HFM in the first simulation.

The experiment in Figure 4, $\underline{h}$ raises to the $\underline{t}$ power is within the limits of 117,649 because the brute -force approach is very time consuming. Hence, we compare HFM wit CBM alone in the next experiment. Figure 7 presents the test patterns of the second experiment. The amount of t asks and hosts listed in Figure 7 is more than that in Figure 4. The results are illustrated in Figure 8. We use Eq.(7) to measure the efficiency of HFM. Similarly, the experiment result shows that CBM performs better than HFM even if tasks and hosts are increased.

```
int[] CP = {15,30,45};          // random seed for CP
int[] WL = {15,30,45};          // random seed for WL
int[] Hosts = {3,4,5,6,7,8,9};  // No. of hosts
int[] Tasks = {10,20,30};               // No. of tasks
ran r = new ran();                      // a user defined class

for (int i = 0;i < CP.length;i++)
        for (int j = 0;j < WL.length;j++)
                for (int m = 0;m < Hosts.length;m++)
                        for (int n = 0;n < Tasks.length;n++)
                        {
                                r.genHost(Hosts[m],CP[i],(5*CP[i])/7); // randomize CP
                                r.genTask(Tasks[n],WL[j],WL[j]/11);   // randomize WL
                                CBM(); // Competence Based Method
                                HFM(); // Heavy First Method
                        }
```

Figure 7. The simulation patterns for independent task graph in the second simulation.

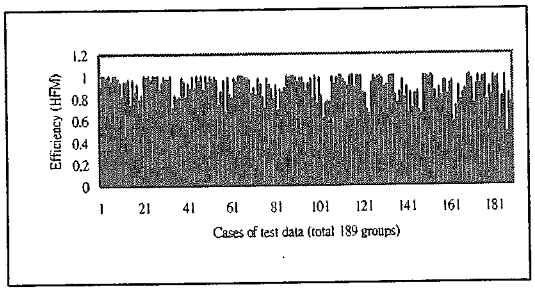

Figure 8. The result of the second simulation.

The efficiency, as we can see for most of test cases, is less one, that means the parallel time for CBM is less than that for HFM. The linchpin of scheduling is the consideration of which host is suitable for the task. As a result, sometimes the CBM defers a task with higher priority, in other words, the CBM would rather make an available host idled. The only disadvantage for CBM is that its scheduling overhead is heavier than that of HFM. The worst case for CBM's time complexity is $O(N^2)$, where N is the number of tasks. It happens when the *Fact.1(b)* is used for all comparisons.

Another experiment is to examine the performance about CBM for the scheduling problem of dependent task graph. In [17], it presented three heuristic methods: Heavy Nod First (HN), Critical Path Method (CPM), and Weighted Length Algorithm (WLA). We study eight DAGs and replace CB M in Eq. (7) by CPM . Figure 9 depicts the experiment parameters. We run 189 cases (3*3*7*3 = 189) for each DAG. Figure 10 and Figure 11 show all DAGs for simulation. These DAGs can be found in the literature. Figure 10-(a), Figure 10-(b), and Figure 10-(c) are example task graphs from [7]; Figure 10-(d) and Figure 11-(e) can be found in [10]; Figure 11-(f) is used in [11-12]; Figure 11-(h) is taken from [9], respectively; we found the same DAG in Figure 11-(g) in [8, 17]. Figure 12 shows the simulation results. We compute the efficiency of 189 conditions and then obtain their average efficiency. As one can see that the average efficiency for CBM, HNF, and WLA are all less than one. It means that the performance of CBM is better than all of CPM, HNF, and WL for dependent task graph scheduling problem.

```
int[] hosts = {15,30,45};                                         // random seed for CP
int[] tasks = {15,30,45};                                         // random seed for WL
int[] basic = {3,4,5,6,7,8,9};                                    // No. of hosts
int taskNum;
ran r = new ran();                                                // a user defined class
main m = new main("data.dag"+args[0]);                            // read a DAG from a file
dagp d = new dagp("./","data.dag"+args[0],dagp.HNF);              // a DAG parser
taskNum = d.size();                                               // No. of tasks
for (int i = 0;i < hosts.length;i++)
        for (int j = 0;j < tasks.length;j++)
                for (int k = 0;k < basic.length;k++)
                        for (int l = 0;l < 3;l++)
                        {
                                r.genHost(basic[k],hosts[i],(5*hosts[i])/7);      // randomize CP
                                r.genTask(taskNum,tasks[j],(tasks[j]/11));       // randomize WL
                                m.HNF();                                          // Heavy Node First
                                m.CBM();                                          // proposed method
                                m.CPM();                                          // Critical Path Method
                                m.WLA();                                          // Weighted Length Algo.
                        }
```

Figure 9. The simulation parameters for DAGs.

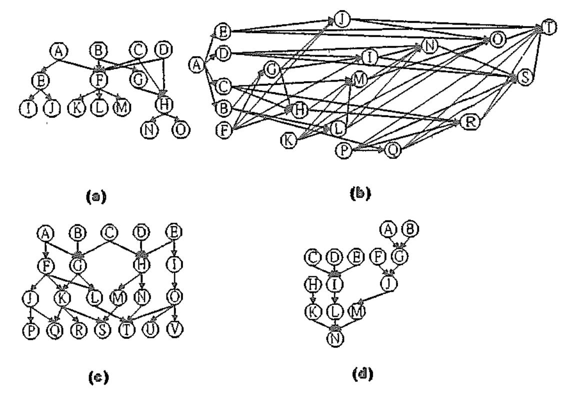

Figure 10. The simulation DAGs (part 1).

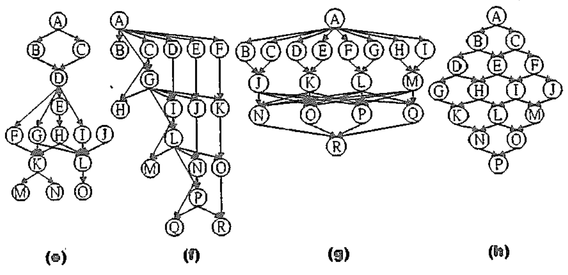

Figure 11. The simulation DAGs (part 2).

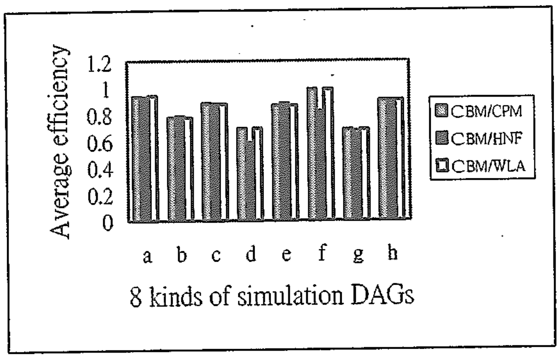

Figure 12. The average efficiency on eight kinds of DAGs.

6. Conclusions

In this paper, we have proposed a heuristic scheduling problem, called Competence Based Method to dynamically distributed workload in a distributed computing environment using Java. It is designed to meet four properties of Web computing: heterogeneity, scalability, centralization, and non -dedication host. CBM befits non-deterministic natures of distributed computing (Property 4 mentioned in Section 1). Compared with the competing algorithms, the simulation results show the proposed CBM performs faster in parallel time and closer to the optimal one for both independent and dependent task scheduling problems.

References

[1] J. Gosling and H. McGilton, " he Java Language Environment," Sun Microsystems, Mountain View, 1995.

[2] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff, "Charlotte: Metacomputing on the Web", In Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems, 1996.

[3] L. Vanhelsuwe, "Create your own supercomputer with Java", JavaWorld, 2(1), 1997.

[4] P. Cappello, B. Christiansen, M. Ionescu, M. O. Neary, K. Schauser, and D. Wu, "Javelin Internet-Based parallel Computing Using Java ", Concurrency: Practice and Experience, Vol. 9, No. 11, pp. 1139-1160, Nov. 1997.

[5] P. A. Gray and Vaidy S. Sunderam, "IceT: Distributed Computing and Java", Concurrency: Practice and Experience, Vol. 9, No. 11, pp. 1161-1167, Nov. 1997.

[6] A. Keren and A. Barak, "Adaptive Placement Of Parallel Java Agents in a Scalable Computing Cluster", In Proceedings of the ACM Workshop on Java for High-performance Network Computing, 1998.

[7] B. Shirazi and M. Wang, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling", Journal of Parallel and Distributed Computing, Vol. 10, pp. 222-223, 1990.

[8] Hesham El-Rewini and T. G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines", Journal of Parallel and Distributed Computing, pp. 138-153, 1990.

[9] D. A. Menasce, D. Saha, S. C. D A Silva Porto,V. A. F. Almeida, and S. K. Tripathi, "Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures", Journal of Parallel and Distributed Computing, pp. 1-18, 1995.

[10] Hesham El-Rewini and Hesham H. Ali, "Task Scheduling in Multiprocessing Systems," IEEE, 0018-9162, 1995.

[11] M. Y. Wu and D. D. Gajski, "Hypertool: A Programming Aid for Message -Passing Systems", IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 3, pp. 330-343, Jul. 1990.

[12] Y. K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 5, May 1996.

[13] H. J. Siegel, J. K. Antonio, R. C. Metzger, and Y. A. Li, "Heterogeneous Computing", Parallel and Distributed Computing Handbook, A. Y. Zomaya, ed., pp. 725-761. New York: McGrawHill, 1996.

[14] Y. Yan, X. Zhang, and Y. Song, "An Effective and Practical Performance Prediction Model for Parallel Computing on Non -dedicated Heterogeneous NOW", Journal of Parallel and distributed Computing, pp. 63-80, 1996.

[15] Pathak, G., and Agrawal, D. P. "Task Division and Multicomputer System", Proc. 5th Int. Conf. On Distributed Computing System, Denver, CO, p.273, May 1985.

[16] Michael Pinedo, "Parallel Machine Models", Scheduling Theory, Algorithms, and Systems, pp. 61-92, New Jersey, Prentice-Hall, 1995.

[17] Tao Yang and A. Gerasoulis, "List Scheduling with and without Communication Delays", Parallel Computing, pp. 1321-1344, 1993.

Appendix I

The Heavy First Method (HFM) Algorithm

```
Input: A set of tasks, $\underline{T} = \{T_1\ T_2 \ldots T_t\}$, and a group of hosts, $\underline{H} = \{H_1\ H_2 \ldots H_t\}$.
Output: A task schedule of $\underline{T}$ on $\underline{H}$.
1.  Initially, every $T_j$ is a member of $\underline{T}_{wait}$.
2.  Use workload and computing power as the priority of tasks and hosts, respectively, and ties are broken arbitrarily.
3.  For each host, repeat 3.1-3.3.
    3.1 Let $H_i$ be the host with highest priority among current available hosts.
    3.2 Let $T_{wait}$ be the task with highest priority in $\underline{T}_{wait}$. Allocate $T_{wait}$ to $H_i$.
    3.3 Move $T_{wait}$ to $\underline{T}_{exec}$ and $H_i$ becomes unavailable.
4.  Whenever a host becomes available, assign it with the highest priority task in $\underline{T}_{wait}$.
```

In this algorithm, load balancing is achieved through assigning the heaviest node first and the similar algorithm can be found in [7, 15-16]. Step 2 may use a max-heap or a sorted list to accomplish and complexity is $O(\underline{t}\ log(\underline{t}) + \underline{h}\ log(\underline{h}))$. Suppose that we use a sorted list for storing those inputs, Step 3 and Step 4 require $\underline{t}$ repetitions totally and the computation complexity of every repetition is about log(1). Thus the total time complexity of HFM is $O(\underline{t}\ log(\underline{t}) + \underline{h}\ log(\underline{h}))$.