A Load Balance Strategy for P2P Networks

MITICĂ CRAUS[1] and CĂTĂLIN BULANCEA[2]

[1] Department of Computer Science an Engineering
Technical University "Gh.Asachi"
Blvd. Dimitrie Mangeron 53A, 700050 Iasi
ROMANIA

[2] Institute for Computer Science
Romanian Academy
Blvd. Carol I 22A, 700505 Iasi
ROMANIA

***Abstract**: -* A new decentralized Load Balance (LB) paradigm for P2P Networks is presented and analyzed. Our strategy is based on a gradient map which is build by a pheromone-like technique inspired from the Ant Colony Optimization metaheuristic. The tasks are assumed to have a certain running time (*T_run*), which is considered to be a random value uniformly distributed in the interval (T_run_{min}, T_run_{max}) and a transfer time *T_trans*, which is also a random variable from the interval (T_transf_{min}, T_transf_{max}). New tasks can appear spontaneously. Each task is intending to wait a certain time in a workstation, time which cannot exceed a predefined value *T_queue*; the probability to request a transfer for another workstation is a random variable uniformly distributed in the interval (0,*T_queue*). *T_queue* parameter is depending of *T_transf*. When the new tasks do not appear in every processing node, LB is efficient.

Key-Words: - load balancing, P2P networks, task migration, gradient map, pheromone trails, Ant Colony Optimization

1 Introduction

Some LB strategies assume centralization. In these strategies there is a watching device which measures the unbalance of the system and when this unbalance exceeds some thresholds, starts the rebalancing procedure (using either classical, evolutionary or GA strategies). These strategies are generally exposed to common drawback in many-to-one communication paradigm, called *bottleneck* [2,6].

The decentralized LB strategies try to reach and maintain a global balance using local neighbor-to-neighbor transfers. They can be classified in either LB based on *global* or *local* knowledge and connectivity. A global method has the advantage of the fast load diffusion, but the communication of type all-to-all can delay the balancing process if there is a big number of processing nodes [5]. The approaches based on local knowledge and connectivity are not affected by the big number of processors and can easy be expanded. This characteristic is called *scalability* [4,6]. One of the main drawbacks of this approach is the slow diffusion rate.

In our model, the LB strategy is decentralized and local. A certain workstation has only local knowledge about the workload. In order to route the tasks, it uses a local routing table based on a *pheromone trail map* with *asymmetric update rules* [2]. A priority queue contains the tasks which intend to run on the workstation which hosts the queue. Always, the task with the biggest transfer time will be in the front and is the first candidate to be performed.

2 Model Description

2.1 Assumptions

In our model, each workstation is regarded as a node *v* of a graph $G=(V,E)$, $V=\{1,2,\ldots,n\}$, $E\subseteq V\times V$.. Such node is called *processing node* or *computing* *node*. Connections are represented by edges. The network topology may be a incomplete graph (a graph with some of the nodes not connected by edges). That means that some nodes may work as gateways between two, three or more sub-graphs, but it is important to mention that all the nodes have the same routing system. This strategy is built only on local knowledge and no global routing systems are required. The tasks have a certain *running time (T_run),* which is a random variable uniformly distributed in the interval *(T_run*$_{\min}$*, T_run*$_{\max}$*)* [2]. It would be important to mention that the running time is not known in advance for the routing and the queuing procedures. There is supposed also that there are no dependencies between tasks. Time costs are also considered for transfers. Each task has a *transfer time (T_transf)*, which was considered as random variable also, uniformly distributed in the interval *(T_transf*$_{\min}$*, T_transf*$_{\max}$*)*. There are also some *additional information* attached during task *lifetime* (the time between its appearance and the start of its execution). This information refers to the number of transfers, the origin node and the task's ID [1,2].

2.2 Queuing system

The queuing system works as it is shown in Fig. 1.

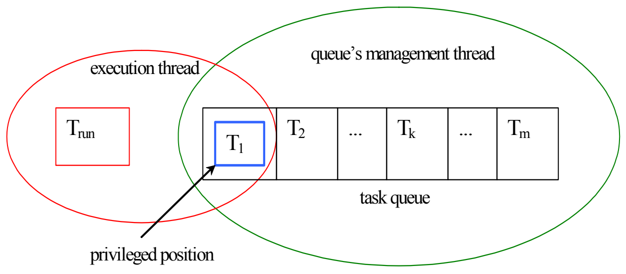

Fig. 1 – The computational logic of the nodes

As it can be seen in figure above, each node has a thread for queue's management. This thread adds or removes tasks in the queue. The tasks are added in order of their appearance or arrival at the node. The implementation of the queue is using the heap structure. After every insertion or deletion, the heap structure of the priority queue is remade. In this way, first position has a special purpose because it hosts the task with the greatest transfer cost from current node

During its lifetime, each task is queued in different nodes and waits to be executed. If it is not executed in a certain time, the task will request transfer to another node. For a certain task *k*, queued in the node *i*, the probability to request the transfer is uniformly distributed in the interval (0*, T_queue*$_k$):

$$p\_req\_transfer_k(t) = \begin{cases} \dfrac{t}{T\_queue_k}, t < T\_queue_k \\ 1, t \geq T\_queue_k \end{cases} \qquad (1)$$

where *p_req_transfer*$_k$*(t)* is task's *k* probability to request to be transferred to another node, *t* is the time spent in current node and *T_queue*$_k$ is the task's *k* time limit for queuing in the current node.

The *T_queue*$_k$ parameter is computed using the following formula:

$$T\_queue_k = qt\_grad * T\_queue_{\min} * \frac{T\_transf_k - T\_transf_{min}}{T\_transf_{\min}} + T\_queue_{\min} \qquad (2)$$

where *T_queue*$_{min}$ is the minimal value of queuing time (is assumed to be the queuing time for the tasks having *T_transf = T_transf*$_{\min}$; this is a common value for all tasks which may appear in the system).

Equation (2) tries to minimize the global time spent for transfers. It can be said that tasks which have expensive transfer cost are encouraged to wait more in nodes.

In order to redirect the transfers to the unexplored subgraphs, a memory system was designed. Considering node *i* is current node for task *k*, let N_i be defined as it follows:

$$N_i = \{ j \mid j \text{ is adjacent to } i \} \cup \{ i \} \qquad (3)$$

If task *k* is deciding to migrate from the node *i* at time *t*, the destination node will be chosen from a routing table using the next procedure [2].

```
if (there are unexplored adjacent nodes)
        choose_dest($N_i^{(k)}$')
else
        choose_dest($N_i^{(k)}$")
```

where:

$$N_i^{(k)}{}' = N_i \setminus \{ j \mid j \text{ was already visited by the task } k \} \qquad (4)$$

$$N_i^{(k)}{}'' = N_i \setminus \{ j \mid \text{task } k \text{ already migrated sometime from } i \text{ to } j \} \qquad (5)$$

2.3 Migration routing

In order to increase the efficiency of task migrations, a modified *ACO pheromone trail metaheuristic* is used for task routing [3,4]. As it was already stated in [1], the trails have *directional attributes* and the routing tables are local; the *normalization* process is also local (∑trails=constant).

The value of the entry *j* of the routing table of node *i* $(entr_{ij})$ will be [3]:

$$entr_{ij} = \frac{[tr_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l\in N_i^*}[tr_{il}(t)]^{\alpha}[\eta_{il}]^{\beta}} \tag{6}$$

where tr_{ij} is the trail corresponding to the edge *ij*, α and β are two parameters that control the relative weight of the pheromone trail and the heuristic value, respectively.[4] In this model, the used heuristic (η_{ij}) could be the inverse of the *j-th* neighbor load .[2]

The role of α and β is stated as it follows: if $\alpha=0$, then the least loaded node is preferred in the selection process. If on the contrary $\beta=0$, the pheromone is the basic selection criterion. Although this method will lead in most of the cases to the rapid emergence of stagnation, in this model this is avoided by using asymmetrical update rules for the trails.

If a task is leaving the node *i* for the node *j* using the *ij* edge, then the following update rule is applied for corresponding trail:

$$tr_{ij}(t+1) \leftarrow tr_{ij}(t) + \delta tr_1 \tag{7}$$

where tr_{ij} is the trail corresponding to the edge *ij*, and δtr_1 is the amount of trail added to trail *ij*.

If a task is coming in the node *i* from the node *j* then the trail *ij* is decreased following the next update scheme:

$$tr_{ij}(t+1) \leftarrow tr_{ij}(t) - \Delta\delta tr_2 \tag{8}$$

where δtr_2 is the amount of trail subtracted from the trail *ij* if a task is coming from *j* to *i*.

As it was stated before, the trail values are normalized during the update process and they cannot take negative values.

In order to increase the efficiency of the task routing operation toward the nodes with low load, we have considered $\delta tr_1 << \delta tr2$. This restriction will also minimize the probability of having ping-pong transfers between nodes.[1] The δtr_1 and δtr_2 have a global characteristics; this will result in uniformly marking transfers around the entire network[2].

For an adjacent node *j* which was not explored previously, the probability to be visited by the task *k* is:

$$p_{ij}^{(k)}(t) = \frac{tr_{ij}(t)}{\sum_{l\in N_i^{(k)\prime}} tr_l(t)} \tag{9}$$

where $N_i^{(k)\prime}$ was defined in (5).

If all the adjacent nodes of *i* were already visited (including node *j*) then the node *j*'s probability to be chosen by the task *k* is:

$$p_{ij}^{(k)}(t) = \frac{tr_{ij}(t)}{\sum_{l\in N_i^{(k)\prime\prime}} tr_l(t)} \tag{10}$$

where $N_i^{(k)\prime\prime}$ was defined in (6).

3 Implementation

The algorithm which implements our model is stated below. The agents encapsulate tasks (one per agent); they store also information about the previous migrations [1].

```
for each node parallel do
  for each agent in current node do
    if (agent.t < agent.T_queue) then
      agent.decide_t()
    else
      agent.decision=leave
    if (agent.decision=leave) then
      agent.choose_node(pheromon_table)
      agent.add_to_memory(current_node)
    else
      agent.update_t()
  end_for
  population.update()
  leave_update_ph_tables(pheromon_tables)
  exchange_tasks()
  arrive_update_ph_tables(pheromon_tables)
  if (agents arrived in current node) then
    population.update()
    for each agent arrived in node do
      agent.reset_t()
end_parallel_for
```

4 Experiments

The experiments were done for architecture of 16 computing modules. The communication topology was represented by a graph G with 16 nodes. In order to simulate the task transfer between computing modules, the MPI library was used. The tests were performed on a Sun HPC service having a backend with 24 processors.

Task's running time was not known in advance for the routing and queuing procedures. This was used only to measure the efficiency of the LB paradigm. A random scheme for simulating the task generation was used. For tests, there were considered three levels of the task density. These will be detailed below.

LB quality was measured using two parameters: the average of the *speedup* and the *relative standard deviation* for workload performed in all graph nodes.

For a task k, the speedup can de defined as:

$$sp_k = \frac{t\_wait_k}{lifetime_k} \tag{11}$$

where:

- $wait_k$ is the the period from the task birth (appearance) until the task running, in the case of the lack of the load balancing strategy. This means that the task does not migrate.
- $lifetime_k =$

$$\sum_{i-\text{visited by task } k} t\_wait\_in\_node_i + migr\_count * t\_transf_k \tag{12}$$

and *migr_count* stores the number of migrations for task k.

The relative standard deviation was defined as:

$$relstdev = \frac{\sigma}{avg\_workload} \tag{13}$$

where:

- σ is the *standard deviation* for workload performed in graph nodes;

$$\sigma = \sqrt{\frac{\sum_{i \in G} (p\_workload_i - avg\_workload)^2}{n-1}} \tag{14}$$

- *avg_workload* is the average value of workload performed in nodes.
- *p_workload$_i$* represents the performed workload on node i
- G is the graph topology used for experiments with n nodes.

The topology used for experiments is shown in Fig. 2.

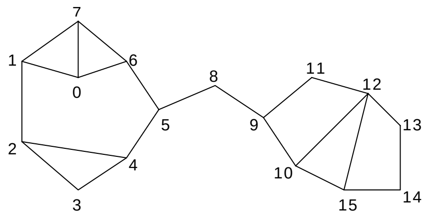

Fig. 2 - Graph topology used for experiments

In the charts bellow the two performance parameters stated before are visualized.

For experiments, it was chosen the next set of constrains:

$$\begin{aligned} &T\_run_{\max} = 0.1 * T\_simulation \\ &T\_transf_{\max} = 0.5 * T\_run_{\max} \\ &T\_transf_{\min} \cong T\_run_{\min} \end{aligned} \tag{15}$$

and *T_simulation* was considered the global time of simulation.

First set of tests was made using a technique of generating tasks inside few computing nodes ($\lfloor n/3 \rfloor$ nodes). Tasks were generated only in nodes having *rank*=3**k*, k integer. The average value of the tasks per experiment was 87.9.

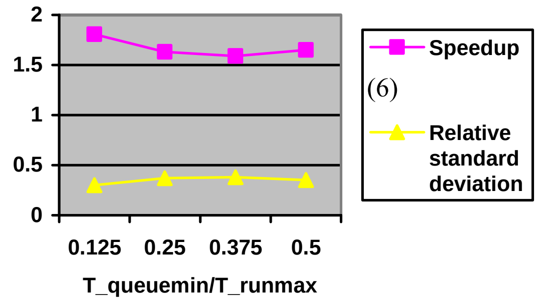

Fig. 3 – Experiments for low density of the tasks appearance

As it can be observed in Fig. 3, in the case of the systems with low density of the tasks appearance, it makes sense to perform transfers intensively. The *average speedup* value is increasing if more transfers are made.

The second set of tests was performed using a generation scheme that works over $\lfloor n/2 \rfloor$ processing nodes. The tasks were generated only in the nodes with an odd rank. The average value of tasks per experiment was 116.4.

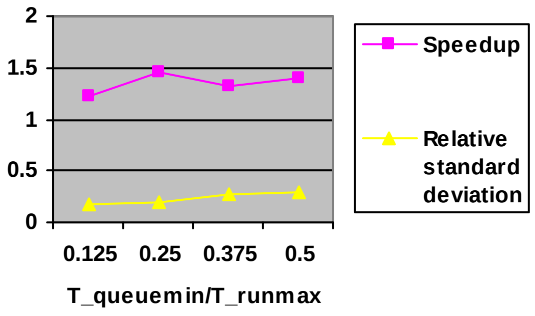

Fig. 4 – Experiments for average density of the task

In systems with average density of the tasks appearance, the LB procedure offers good *average speed-up,* but a high transfer rate affects negatively the LB performance.

Third set of tests was made using a procedure of generation tasks inside every processing node. The average value of tasks per experiment was 242.3.

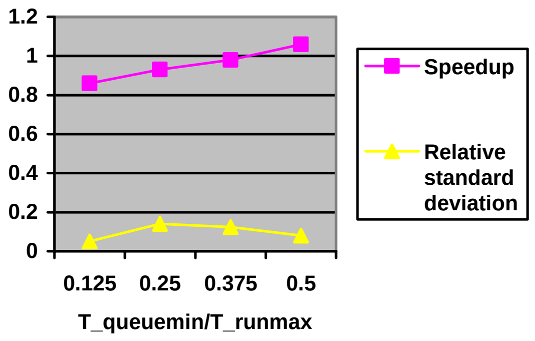

Fig. 5 – Experiments for high density of the tasks appearance

As it can be seen in Fig. 5, in the case of high density of the tasks appearance, a high transfer rate makes the LB process inefficient. To solve this situation, is better to use a procedure which sends with high accuracy the tasks from overloaded to underloaded nodes. However, a small transfer rate is showing some improvement.

5 Conclusions and Future Work

Applying our LB paradigm, the load of a distributed system evolves continuum to a uniform distribution, without blocking the activity in order to balance the load. Also, is not necessary to accumulate information about an eventually unbalancing.

As it was observed in the experiments, the performance of our LB strategy is depending on the way that the tasks may appear in the system. When new tasks do not appear in every processing node, LB is efficient.

The future work will be focused on improving the adaptive capacity of our LB paradigm by adding a *local self optimizing* method for T_queue_{min} parameter. The simulation time will be split in phases and after each phase, a *self-evaluating* procedure will be performed for each node. The T_queue_{min} parameter will be adjusted locally, depending on the each node's efficiency for previous phase.

6 Credits

The authors would also like to acknowledge the support of the Romanian HPC Centre "CoLaborator".

References

[1] M. Craus and C. Bulancea, An agent based model for real-time load balancing in non-uniform connected computing environments, In *Intelligent Systems - Selected papers from ECIT 2004,* Iasi, Romania, 2004, pp.239-250.

[2] M. Craus and C. Bulancea, A decentralized Load-Balancing strategy using gradient maps", in *Intelligent Systems - Selected papers from SASM 2005,* Iaşi, Romania, 2005, pp. 79-88.

[3] M. Dorigo and G. Di Caro, The Ant Colony Optimization Meta-Heuristic, *New Ideas in Optimization*, D. Corne, M. Dorigo and F. Glover (eds), McGraw-Hill, 11-32.

[4] M. Dorigo, G. Di Caro and L. M. Gambardella, Ant Algorithms for discrete optimization, *Artificial Life*, Vol. 5, No. 3, 1999, pp. 137-172.

[5] D. Grigoras, *Parallel Computing – Systems and Applications*, Computer Libris Agora, 2000, Cluj Napoca, Romania, pp. 338-360.

[6] D. Heinrich, The Liquid Model Load Balancing Method, *Journal of Parallel Algorithms and Applications*, Special issue on Algorithms for enhanced Mesh Architectures, Karlsruhe University,1996.