Informed Load Balancing

Shengda Hu

December 20, 2000

1 Introduction

In distributed computing, the progress of a job is determined by the slowest server that is working on it. To see a well paced progress, we need all the servers proceed at a reasonably matching manner. An overloaded server would not provide reasonable performance, while an underloaded server is not doing what it's supposed to do. In this view, it's important to balance the load on the servers. There can be various schemes of attacking the problem. Here we'll try to explore the dynamic approach. In order to balance the load online, clients need to know about what is happening on the servers, and servers need to know what is going on on the client side, i.e. we need to get some information across the server-client boundary. In this paper, we are concerned on how to request for information, and how to send it.

2 Setup

In full generality, the system can be described as a graph, with a workstation at each node to be either server or client. The data flow in this graph has three parts: client-client communication, server-server communication, client-server communication. The first two parts are not really relevent to what we want to consider. The third part contains execution requests, status checking, return of results, etc. In order for client to know anything about the servers, the last part of the data flow is the place to look at. In many cases, things will be much simpler if there were a name server, with which everybody registers. The name server would be used when starting a job, to establish the connection of client to the servers, and setup all the necessary parameters. Afterwards, name server should be used as little as possible to reduce the possibility that it becomes the bottleneck.

We'll consider a much simpler and naive setup. There is only one client machine, and it sends tasks to many server machines. In this case, the servers do not need to track anything from the client, so the only information issue is client getting information about servers. There is no name server, so the client talks directly to the servers. The remote execution is also carried out in a most naive way. The client sends task to servers, and server will send back a acknowledge message when it receives the execution request. After a task is completed, server sends the return status back to the client. Although it's desirable to have something like RPC, which make sure that the task sent to server is executed, we opted not to worry about it for simplicity. We further simplify the matter by assuming that the transfer of execution request and result of execution involves only small size messaging and no data transfer.

3 Schemes

The schemes of requesting and sending information can be classified into four classes, as suggested in [**?**]. They are None, Implicit, Parasitic, and Explicit. None do not utilize any information. Implicit observes the traffic between server and client, and deduce useful information. Parasitic scheme attachs the request or information to existing traffic. Explicit scheme will send extra messages to request for or send out information. Server and client do not have to use the same scheme.

In current setup, client using implicit scheme means to keep track on how many outstanding jobs are on each server, while server using implicit scheme is basically the same as using none scheme, since there is only one client. For parasitic schemes, it means that machines attach request of information and load information to the message flow of the underlying remote execution layer. For explicit schemes, the load information is requested when the client has tasks to distribute, and it'll wait for load information to flow back, then distribute tasks accroding to updated information.

The least amount of extra work the servers and clients should do using various schemes are shown in the table below:

table 1

Scheme	**Server**	**Client**
None	none	none
Implicit	Keep track of outstanding tasks from each client	Keep track of outstanding tasks at each server
Parasitic	Attach request of information to outgoing messages	Attach information to outgoing messages
Explicit	Send request of information when needed	Send information when requested

When using Parasitic scheme, the other side will have to extract the request or information from the message, which is of cause extra work.

The following table compares the effect of different scheme on the traffic it caused and the quantity as well as quality of information it can get.

table 2

Scheme	**Traffic**	**Information**
None	no extra traffic	knows nothing
Implicit	no extra traffic, no extra bytes	limited knowledge, in this case only outstanding task number
Parasitic	no extra traffic, extra bytes	better knowledge information could not be timely
Explicit	extra traffic, extra bytes	more timely information

We are going to test which combinations of schemes work better on the current setting. We'd like to point out that the None and Implicit schemes for the servers are actually the same, since there is only one client. In general, those two should be different, where server should keep track of the outstanding tasks from each client when using Implicit scheme, and accepts or refuses execution request from a client according to the number of outstanding tasks.

4 Implementation

We designed the mechanisms for information layer with Parasitic scheme in mind, since that is the most complicated scheme as to implementation. In order for server to keep track of load information, and for client to utilize load information, we added some data structures. Then we provided an interface for application programming.

4.1 Data structure: Client side

The client keeps a list of servers that it connects to, and assigns to each host a numerical priority according to the information it has at the time. When the client distribute tasks to server, it will check the list to see which host has highest priority, and send it the task. The priority of the host is updated when there is information arrived for that server in parasitic or explicit schemes, or when task is distributed to or completed by the server when using implicit scheme. The list is sorted with respect to the priorities when distributing tasks to the servers, and pick the highest priority (largest numerical value) ones to give tasks to.

In order for greater flexibility, application can specify what kind of information it needs, as well as the function that converts load information to numerical priority.

4.2 Data structure: Server side

The server stores load information in a circular queue. The server checks load information periodically, and store it in the current position of the queue, which simply overrides the old information. It can be done differently, e.g., check load information only when preparing for sending out. Our choice has the potential advantage of keeping longer history, which should be useful; while the drawback is the time granularity can not be smaller than the time interval which the server checks load information.

In full complexity, the server should also keep a list of clients that connects to it, and keep track of the number of outstanding jobs, or request for information from each client. In current setup, those are not necessary.

4.3 Data structure: information representation

Although we only worked with cpu load when we test this system, we designed the information represtation scheme with other kinds of "load" in mind.

Server gets load information from the `/proc` pseudo-filesystem in Linux. The file `stat` contains most of the information such as cpu usage, disk usage, paging, and swapping, etc. Also in the same directory, the files `meminfo` and `loadavg` contains respectively memory usage and avrage length of running queue.

Client requests for information by sending over an integer `rtype` whose bit pattern tells the server what kind of information it requests. The server keeps an integer `itype` whose bit pattern represents which load information it's willing to reveal. When server gets request from client, it will first compute `(rtype & itype)` to see what can be sent to client. Then server will check the circular queue and send the information over with the value `(rtype & itype)`.

4.4 Messaging interface

The messaging interface is a internal interface, between the information layer and remote execution layer. There are four methods to attach (or extract) request and load information onto (resp. from) typed messages. When calling attach methods, if the message is `NULL`, it packs request or information into messages.

- `VILIB_attach(extract)InfoRequest(Message *m, ...);`
- `VILIB_attach(extract)HostInfo(Message *m, ...);`

4.5 Client interface

The interface to client provides methods to get information from a list of hosts, and select hosts from a list of hosts. Client can define a age threashold such that information older than the threashold will be considered stale, and can then carry out proper action. So, there is a method to check which hosts has stale load information.

- `Vlib_GetInfo(prilist_t *hostlist);`
- `Vlib_GetRandom(prilist_t *hostlist, int n);`
- `Vlib_PickHosts(prilist_t *hostlist, int n);`
- `Vlib_CheckStale(prilist_t *hostlist, int threashold);`

4.6 Server interface

The interface to server is not that strigent as the one for client. The main reason is it's not complete, in terms of the ability to implement all the schemes. It only enables servers to attach load information to messages when sendinng them out, and also enables server to check the load information that it's willing to reveal. Since the current setup only has one client, the server interface do not need those calls such as those for picking which client it wants to receive tasks from, etc. This will be a major part to work on if we want to make this system support multiple clients.

- `VILIB_SendReply(..., inforeq_t *req, load_t *load, Message *m, int prepend);`

  When `prepend` is non-zero, the load information `load` is attached to the message `m`, according to the request from client `req`.

- `VILIB_hostCheckInfo(..., itype_t itype);`

4.7 Client schemes

Client keeps track of how many tasks have been sent out and not completed yet. It sets a upper limit of total number of outstanding tasks. When the limit is reached, client will wait for a task to return in order to sent one out. Client will also keep track of number of outstanding tasks on each server. When using none scheme, client will not consult the information when distribute job. When using implicit scheme, client will only check the outstanding task number.

In parasitic and explicit schemes, client will update the priority list when information arrives from server, and will sort the list when distributing tasks, and pick the ones that have highest priority. When using parasitic scheme, when the priority of a server becomes very low, say the lowest possible, then it's not going to get any task if any other server has any priority at all. In that case, without dealing with it, the unlucky server will get starved since it would not be able to update it's priority, and the system could potentially die off one by one. So client set a limit on the age of information, and ignore all the information older than the limit. It will either randomly pick from the servers that have stale information and give a task to it, so that the information could be updated, or send explicit request to the server, when the server's information becomes stale, and then update the information, which is a hybrid of explicit and parasitic schemes. We'll first examine the first implementaion.

For explicit scheme, client can either ask for information from every server or it can randomly pick a subset and send request to them and distribute tasks according to the returned informatio. We are wroking with the later choice. The former choice takes the risk of waiting every server to return information, which might be very costly if there are many servers.

4.8 Server scheme

None and implicit schemes on server side are actually the same, as we pointed out earlier. The parasitic and explicit schemes just add more complexity when dealing with messages, and do not involve any policy decision.

5 Discussion

We did experiments with the system using simulation. The tasks are spin for fixed amount of time on CPU then exit. There are 4 servers, and one task only requires one of them. We used two kinds of workloads, which are:

- Identical tasks seperated by time intervals of same length,
- Random tasks weighted either to the heavy side or to the light side, still, they are seperated by time intervals time interval of same length.

There are 1000 tasks in each of the workloads we experimented with. In the first kind of workloads, we spin for 1 sec, 3 sec, 6 sec, 8 sec, 10 sec, 16 sec, and 20 sec. For 1 sec and 3 sec, the time interval that seperates tasks is 1 second, while all others have that time interval 4 seconds. In the second kind of workloads, we randomize the spin time between 1 sec to 10 sec, and 1 sec to 20 sec, with 80% of the tasks closer to one end of the spectum than 40% of the difference, while the other 20% are closer to the other end than 10% of the difference. The time interval that seperates the tasks are all 4 seconds for the second kind of load. In all, there are 11 workloads. We'll refer to them as 1sec, 3sec, etc, for the first kind, and refer to those weighted more to the heavy end as high10, high20, while the ones that weighted more to the light end as low10, low20.

First, we observe that passing request and information incurs messaging overhead. The time overheads of dealing with attaching and extracting are typically very small (on client side, averages in the range of $0.01 \sim 0.5$ ms, with standard deviation less than 0.1 ms). The bytes overheads of the schemes are actually fixed. When using none or implicit scheme, there is no byte overhead, since no request or information will be sent across the network. For the other two schemes, for each task, there is exactly one message that could contain information request from the client, and depends on parasitic or explicit scheme on the server side, there would be two or one message that could contain information from server. The table below shows the overhead on client side in terms of time and bytes:

table 3. Client side time and bytes overhead

Combination	Client	Server	Average*	Standard deviation **	Bytes
n-n	NONE	NONE	0.01	0	0
i-n	IMPLICIT	NONE	0.01	0	0
p-p	PARASITIC	PARASITIC	0.18	0.035	78
e-p	EXPLICIT	PARASITIC	0.30	0.037	188
p-e	PARASITIC	EXPLICIT	0.08	0.015	204
e-e	EXPLICIT	EXPLICIT	0.15	0.020	440

*: the average of time overhead caused by information passing.
**: the standard deviation of the time overhead.

Next, we look at the average delay and the average percentage delay of the 11 workloads under the six combination of schemes, as shown in the figures (figure 1 and figure 2).

Figure 1: Average Delay

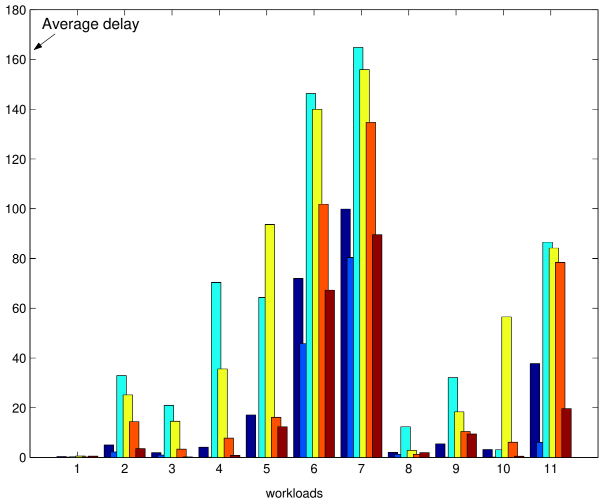
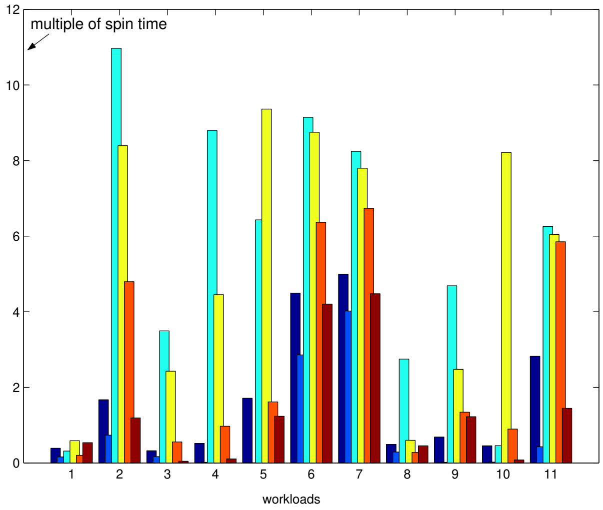


Figure 2: Average Percentage Delay



a) The horizontal axis are the workloads, from left to right are:
1sec, 3sec, 6sec, 8sec, 10sec, 16sec, 20sec, low10, low20, high10, high20.
b) In each group, different colored bars represent different scheme combinations,
from left to right are: n-n, i-n, p-p, p-e, e-p, e-e.

From the figures, we see that n-n combination is actually not that bad, and most of the time i-n combination outperforms all the rest. We think two factors may lead to the phenominon. First, when the experiments were run, the server machines were almost exclusively used for the these experiments, so there was not much outside perturbance which would complicate the situation and made those two unstable. Second, most of the workloads are very regular. The client side distribute tasks in a first-come-first-server manner, which just happened to be great for these regular workloads.

We also see that the combinations using parasitic scheme on client side almost always perform worse than others. We propose the reason to be the untimely-ness of the information client can get in using parasitic scheme. In parasitic scheme, the client send information request along with execution request, so the execution is BEFORE the collection of information. Although the information gets updated, but it's more or less useless.

Another observation is that in the first 7 workloads, the longer the spin time, the worse the best performance of the scheme combination. We suspect the reason to be how the tasks are distributed. The client only keeps an upper limit of overall outstanding tasks, which will in turn give an upper limit to the number of outstanding tasks on the servers. As long as the upper limit on the client side is not exceeded, the client will pick a server and hand it one more task. When the limit is reached, client has to wait for a task to complete in order to distribute one. When tasks are long, the servers would be more likely to get overloaded well before the client side limit were reached, which leads to worse performance. To compensate that, it'd be desirable to put a upper limit not only on overall outstading task number, but also on each server, which essentially means to utilize the implicit information all the time.

Next, we try to examine the hybrid approach to the parasitic scheme, and see if it works better than the pure one. We experimented it using workloads 3sec, 6sec, 16sec, low20 and high20, and scheme combination p-e. We get the following figure 3 and 4:

Figure 1: Comprision of Average Delay

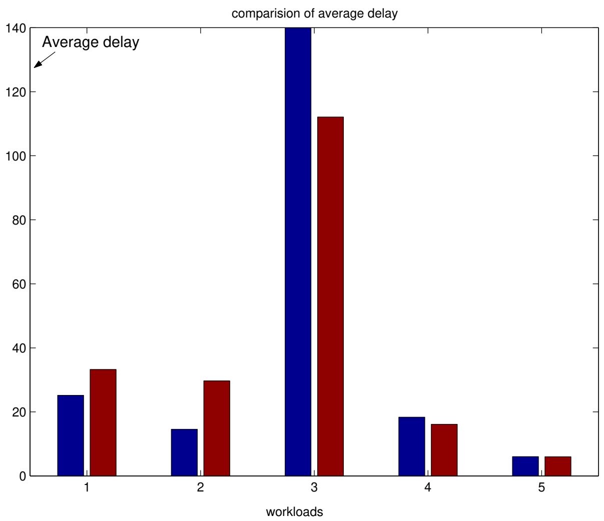
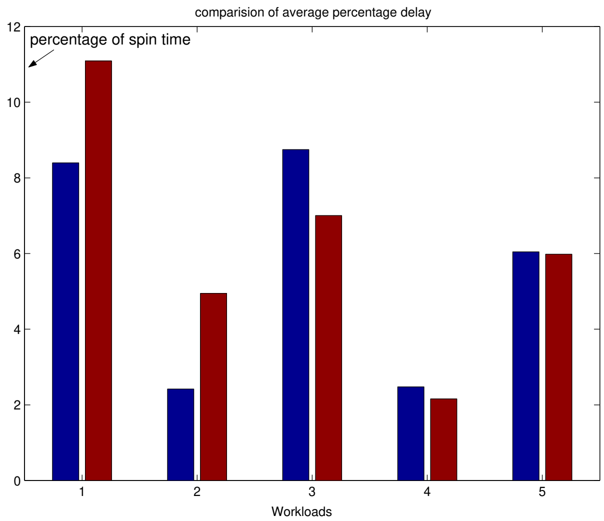


Figure 2: Comparision of Average Percentage Delay



a) The horizontal axis are the workloads, from left to right are:
3sec, 6sec, 16sec, low20, high20.
b) In each group, the left bar is the "old" parasitic scheme,
the right bar is the "hybrid" parasitic scheme.

We see that the hybrid approach performs better when task size increases. We think the reason is that the client will incur more communication under the hybrid approach. When the tasks are small, the gain from better distribution can't match the lose on the communication overhead. When the tasks are longer, the decision at distribution time becomes more important, and the hybrid one catches up.

6 Conclusion

It's said that "stale information is worse than no information", which is shown through the experiments here. Also, we see from the tweak on the hybrid parasitic scheme, that at times it might be worthwhile to consider trading overheads to get better performance. It is also another example that hybrid approach would work better than single one.

7 Future Work

As to the current system, we'd like to see how it behaves under a more realistic workload. For example, one that is taken from the trace of a running system. Also, there are many parameters in the system and more fine tuning to be done, like how to fix the upper limit of outstanding tasks, how long to take information on the server side, etc.

The obvious next step is to have multiple client machines, and see what is the impact of servers knowing information about the clients, and the interplay of policies on both sides. When doing that, it should be important to use a name server.

As we pointed out before, the issue of scheduling both at client side and server side is another interesting point. Right now, the scheduling on client side is the simple first-come-first-serve. In reality, the order of tasks being distributed should satisfy certain condition, which put constraints on which task should go first. It would be interesting to see how to combine the information knowledge with those constraints. We haven't done anything concerning to the scheduling of the server side, nor the cooperation of the scheduling of the both sides.

References

[1] Remzi H. Arpaci-Dusseau: *Performance Availability for Networks of Workstation* Doctoral Dissertation.