Load Balancing in a Cluster Computer

Paul Werstein, Hailing Situ and Zhiyi Huang
Department of Computer Science
University of Otago
Dunedin, New Zealand
werstein, hsitu, hzy@cs.otago.ac.nz

Abstract

This paper proposes a load balancing algorithm for distributed use of a cluster computer. It uses load information including CPU queue length, CPU utilisation, memory utilisation and network traffic to decide the load of each node. This algorithm is compared to an algorithm using only the CPU queue length. The performance evaluation results show that the proposed algorithm performs well.

1 Introduction

A cluster computer is a collection of computers interconnected via some network technology. The individual computers can be PCs or workstations. Ideally, a cluster works as an integrated computing resource and has a single system image spanning all its nodes. Hence, the users see only a single system. User processes can be executed on any node of the cluster.

A cluster can be used for scientific applications that need supercomputing power and in domains such as databases, web service and multimedia, which have diverse Quality-of-Service (QoS) demands. In addition, users can access any node within the cluster and run different types of applications simultaneously. The main goals are to minimise the total response time and maximize throughput.

Load balancing tries to balance the total cluster system load by transferring or starting processes on the more lightly loaded nodes in preference to heavily loaded nodes. Doing so attempts to ensure good overall performance.

The control of a cluster can be centralized and distributed.

- In a centralized cluster, all users interact with the cluster through a central node. The other nodes are processing nodes. User processes are allocated to processing nodes by the central node. The central node collects system state information and makes all scheduling decisions.
- In a distributed cluster, a user can connect directly to any one of the cluster nodes. There is no master node. Each node is considered a local controller. They run asynchronously and concurrently to each other. Each node is responsible for making scheduling decisions for the processes submitted by its users and for accepting remote processes.

A variety of application types can be run on a cluster. Examples include CPU-bound, I/O-bound and memory-bound applications. The different uses of a cluster have different load balancing requirements.

In this research, a load balancing algorithm is developed and its performance is evaluated. The algorithm is based on a distributed cluster where each node makes its own scheduling decisions. A mix of applications is used, including CPU-bound, network-bound, and memory-bound applications.

The performance of the proposed load balancing algorithm is compared to the typical CPU queue length technique used in many other algorithms. The performance tests show the proposed algorithm gives much better throughput.

The rest of the paper is organized as follows. Section 2 presents the current state of load balancing research. A detailed description of the proposed algorithm is presented in Section 3. Section 4 discusses the implementation issues. The experiments and analysis are shown in Section 5. Section 6 gives conclusions and suggestions for future work.

2 Related Work

Load balancing algorithms generally can be classified as either static or dynamic. Static algorithms only use given process and node information in making load balancing decisions. They do not adapt to fluctuations of workload. Under a situation where the system workload is statically bal-

0-7695-2736-1/06 $20.00 © 2006 IEEE

anced, some computers may be heavily loaded while others are idle or lightly loaded.

Dynamic load balancing attempts to balance the workload dynamically by responding to the current system state. It should be able to improve cluster performance. However dynamic algorithms are usually more complex than static algorithms. They collect information on the current cluster state and make decisions on process allocation based on that information. They are more suited to heterogeneous applications. Dynamic load balancing algorithms are based on:

- load estimation policies
- information exchange policies
- process transfer policies
- selection policies
- location policies

Each of these policies is discussed below.

Load information serves as one of the most fundamental elements in the load balancing process. Every dynamic load balancing algorithm is based on some type of load information. In order to balance the workload among nodes, a key issue is the measurement of workload on the nodes. Many types of load information have been explicitly or implicitly used to express the load existing on a node at a given time. Examples include CPU utilisation [6], the length of the CPU queue [4], resource queue length [2] and a method with prediction of process requirements [1].

Once the load information has been collected and analysed according to the load estimation policy, these data must be exchanged among the nodes of the cluster. Information exchange policies include:

- Periodic policies — In a periodic policy, all nodes broadcast their state information at certain time intervals. Either a central node or all other nodes receive and store these data. It is simple and requires no overhead to check the current state.
- Demand-driven policies — Under these policies, a node collects the state of other nodes only when it becomes overloaded or underloaded.
- State-change-driven policies — Under these policies such as [3], nodes disseminate information about their state whenever their state changes by a certain amount.

After the load data are exchanged, a central node or each node may have the load information for some or all nodes. Using these data, a process transfer policy can decide whether a node is lightly or heavily loaded. Most existing process transfer policies are based on various types of thresholds. Different load balancing algorithms use different load information to define the threshold values. Process transfer policies can be based on predefined or dynamically changing single-threshold or a double-threshold values. The process transfer policy determines the load on a node and decides whether to transfer a process.

A selection policy decides which process is selected for transfer. The chosen process could be a new process which has not started or an old process which is already running. Transferring a running process is complicated and not commonly done.

The next step is to select the destination node which is done under a location policy. The node selected should be lightly loaded and have the correct environment to run the process. Shirazi et al. [5] summarized two general policies:

- Minimum load – select a node with the minimum current load.
- Low load – select the first node whose load is below some threshold value. This policy is applied to a transfer policy based on thresholds. There is a possible problem of several heavily loaded nodes transferring their processes to an lightly loaded node, causing it to become heavily loaded. A simple solution is to randomly select one of the lightly loaded nodes for transfer.

Although many policies exist, the policies should be decided according to the desired environment, such as application types or cluster environment. It is very difficult to say which algorithms are most efficient. There is no single algorithm which is optimal for all purposes. We can only find a best solution for a particular situation.

In most clusters, processes will arrive randomly, and it is difficult to know their characteristics such as execution time. We can only take into account the current states of the nodes such as CPU utilisation and CPU queue length.

3 Algorithm Description

3.1 Introduction

The goal of this research is to design an effective load balancing algorithm for a cluster computer. A homogeneous environment is assumed. The individual computers are PCs having a single CPU and their own memory. They are interconnected through switched Ethernet. Each node runs the Linux operating system. The file system is shared through NFS. Users can directly log into any node in the cluster.

The cluster provides network services. User can remotely access web servers, database servers, network printers and also other cluster nodes. Each computer uses the

NFS file server. Therefore network traffic is an important factor in the cluster.

The applications of the cluster are typical Linux applications: the processes can be CPU-bound, network-bound or memory-bound. The type of a new process is unknown. The use of cluster is totally distributed.

The algorithm is decentralized to avoid bottlenecks and a single point of failure. It considers CPU queue length, CPU utilisation, network traffic, and memory usage. These data are exchanged periodically between the cluster nodes. Each node makes its own decisions.

3.2 Load estimation and information exchange policy

Ideally, the load information should reflect the current CPU utilisation, memory utilisation and network traffic of a node. Traditionally, the load of a node at given time was described simply by CPU queue length.

CPU queue length refers to the number of processes which are either executing or waiting to be executed. The processes which are waiting for other system resources are not included. So the CPU queue length does not reflect directly network traffic and memory utilisation. In the proposed algorithm, CPU utilisation, CPU queue length, network traffic and memory utilisation are used.

The system statistics such as CPU utilisation, CPU queue length and the network traffic of a node changes during the life of processes. For example, the CPU utilisation may be high in one second but low in the next second. Therefore it is reasonable to average these statistics over several seconds.

In the proposed algorithm, 5 seconds is set for the averaging interval. CPU utilisation (cpu_u), CPU queue length (nr), memory utilisation (mem_u) and network traffic (net_u) are considered as load information parameters to measure load of a node. The following equation is used to calculate each metric.

$$l_n(par) = \frac{p_1 + p_2 + \cdots + p_t}{t}, \qquad (1)$$

where

- l_n is the average load metric of the specified parameter over the previous t seconds for a particular node.
- par is the information parameter of load. (par is either nr, cpu_u, mem_u or net_u).
- $p_1 \cdots p_t$ is the value of a given parameter in a previous one second interval.
- t is the number of time intervals. t is set to 5 for this research.
- n is the number of a given node.

The averaged information including CPU utilisation, CPU queue length, memory utilisation and network traffic are the load metrics used to describe the load on a node.

The information exchange policy chosen for this research is a periodic policy with a time interval of one second.

3.3 Process transfer policy

In the proposed algorithm, this determination includes two steps. First the nodes are classified according to their loading metrics. Then a decision is made whether to start a process on another node.

3.3.1 Load classification

The first step in the process transfer determination is to classify the load at each of the nodes. The proposed algorithm uses four levels of load: idle, low, normal and high.

The first part of this step involves calculating two threshold values for the CPU queue length (nr), CPU utilisation (cpu_u) and network traffic (net_u). Two thresholds are used because they give a more stable load balancing solution. That is nodes are less likely to constantly switch between highly loaded and lightly loaded.

The calculation of the threshold for these three parameters is done as follow:

1. Calculate load average of each parameter (nr, cpu_u, and net_u) over all nodes. The equation is:

$$L_{avg}(par) = \frac{l_1 + l_2 + \cdots + l_n}{n}, \qquad (2)$$

where

- L_{avg} is the average load of a given parameter over all nodes.
- par is the parameter of load: nr, cpu_u, and net_u.
- $l_1, \cdots, l_n$ are the current load of the parameter of each node derived by load estimation policy.
- n is the number of nodes.

Each of the l_i is the five second average of the desired parameter. See Equation 1.

2. Calculate the threshold values

The upper and lower threshold values of CPU queue length, CPU utilisation and network traffic are calculated by multiplying the average load of each parameter and a constant value.

$$t_H = H * L_{avg}$$

$$t_L = L * L_{avg}$$

where t_H is the high threshold, t_L is the low threshold, H and L are constants. H is greater than one and L is less than one. In the proposed algorithm, H and L are set to 1.3 and 0.7, respectively. That means when a certain load parameter is 30% above the average load, it is highly loaded. When a certain load parameter is 70% of the average load, it is lowly loaded; otherwise, it is normally loaded. Note that these classifications are for individual parameters, not for the nodes.

This calculation and classification of parameters is done for the CPU queue length, CPU utilisation and network traffic. For memory utilisation, the proposed algorithm simply uses the actual five second average. No threshold is applied to it.

The second part of load classification is to group the nodes into one of four classes. Using the threshold values of each parameter, the nodes will be grouped as idle, low, normal or high according to the following criteria. For each node, the CPU utilisation, CPU queue length, memory utilisation and network traffic will be checked to decide whether it is in idle, high, low or normal level.

$$load = \begin{cases} idle & \text{cpu\_u} \leq 1\% \\ high & (\text{mem\_u} > 85\%) \text{ or (cpu\_u is high and net\_u is high) or (nr is high)} \\ low & \text{(cpu\_u is low and net\_u is low) or (nr is low)} \\ normal & \text{otherwise} \end{cases}$$

- Idle – As noted earlier, using only the CPU queue length is not enough to determine whether a node is idle. We use CPU utilisation to determine idle nodes. In a modern network system, there are some background programs running such as daemons or monitoring programs. Most of background programs run periodically for a very short period of time. We set the CPU utilisation to 1% to ignore these background programs. When the CPU utilisation of a node is less and equal than 1%, the node is considered as an idle node.

- High – If any of these three conditions is true, the node will be classified in the high state.

  - Memory utilisation is greater than 85%. This is a characteristic of an individual node, not a cluster average. The proposed algorithm considers any node with less than 15% available memory at risk of being forced into memory paging. Such paging will dramatically slow down processing at that node.

  - Cpu_u is high **and** net_u is high. If both the CPU utilisation is high and the network traffic is high, the node is classified in the high state.

  - Nr is high. If the number of processes in the CPU queue is high, the node is classified in the high state.

- Low – If either of these two conditions is true, the node will be classified in the low state.

  - Cpu_u is low **and** net_u is low. If both the CPU utilisation is low and the network traffic is low, the node is classified in the low state.

  - Nr is low. If the number of processes in the CPU queue is low, the node is classified in the low state.

- Normal – Nodes that are not classified into one of the other categories are the nodes in the normal load state. They are considered as more loaded than the low state and less loaded than the high state.

3.3.2 Transfer decision

After the load of each node has been classified, the next step of the process transfer policy is to decide if a newly arriving process should be run locally or on some other node. The following pseudocode defines how this decision is made.

```
IF the local host is idle THEN
   run locally
ELSE IF there are idle nodes THEN
   run on an idle node
ELSE IF the local host is high loaded AND
        there are low loaded nodes THEN
   run on a low loaded node
ELSE
  run locally
ENDIF
```

This pseudocode gives preference to running a process locally if the local node is idle. The next choice is any other idle node. The next choice is a node with a low load level if the local node is highly loaded. If no node can be found in the previous choices, the process is assigned to the local host.

3.4 Selection policy

The proposed algorithm does not attempt to determine the resource requirements of a newly arrived process. In addition, the proposed algorithm is nonpreemptive. That is, running processes are not moved.

3.5 Location policy

When a process has been selected for transfer, the location policy determines the node to which the process should be transferred. For the cluster used in this research, the arrival of new processes is not centrally controlled and can occur at any time. In addition, there is no communication between nodes when they start new processes locally or start new processes on other nodes. The only communication for the load balancing algorithm is the periodic exchange of load data.

Since all nodes are grouped by the same algorithm, it is possible under worst case conditions for the same target node to be picked by several nodes at the same time. To help avoid this possibility, the proposed algorithm randomly picks a target node from the group of possible targets.

4 Implementation

4.1 Load balancing system architecture

Figure 1 shows the architecture of the load balancing system. LocaBUS [7] provides a very efficient kernel to kernel message passing system for cluster computers. It provides a reliable broadcast function based on Ethernet's broadcast capabilities. LocaBUS reduces the overhead of information collection as compared to using TCP/IP.

LocaMonitor is a distributed cluster monitoring protocol based on LocaBUS. It is implemented as a kernel module and uses locaBUS as the message passing system. It performs the following six tasks:

- Load information collection – Every second, LocaMonitor on each node gathers CPU utilisation, CPU queue length, memory utilisation and the network traffic from the kernel data structures and broadcasts the data clusterwide.
- Calculates a running five second average of the broadcast data received.
- Averages each parameter for all cluster nodes.
- Calculates the dynamic threshold values for CPU utilisation, CPU queue length and network traffic.
- Classifies nodes into different load levels.

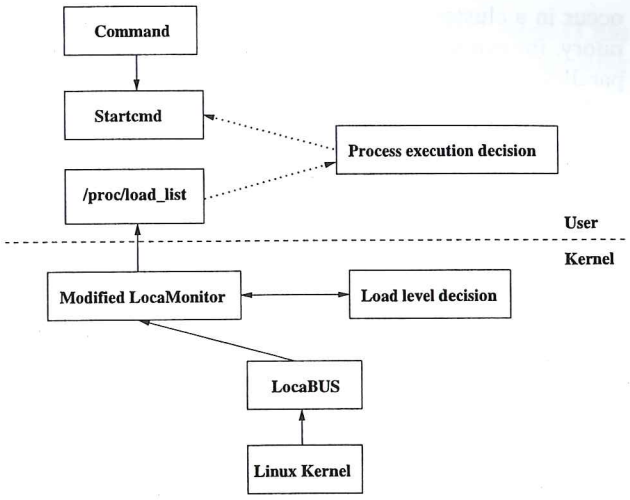

Figure 1. Load balancing system

- Reports the results to /proc/load_list.

When a user enters a command to start a new process, the command is intercepted by `Startcmd` as shown in Figure 1. `Startcmd` is implemented in user space. Its task is to determine which node should run the new process and then to start the process on that node.

5 Experiment

The proposed load balancing algorithm is based on CPU utilisation, CPU queue length, memory utilisation, and network traffic. It is compared to the traditional CPU queue length based policy. This allows a comparison between the two load estimation policies.

The performance tests use a variety of different types of applications: CPU bound, network bound, memory bound, and mixed applications.

All nodes in the cluster are homogeneous and have the same hardware and operating system. The network is switched 100 Mbps Ethernet. The filesystem is NFS with the disks on node01.

The comparisons are based on three aspects:

- Explore which policy is better at finding idle nodes.
- Explore the results of the two policies for certain types of applications.
- Explore the results of the two policies for mixed types of applications.

The experiments are done based on a variety of applications. These applications are meant to simulate what might

occur in a cluster that is used by a computer science laboratory, for example. The tests are not meant for simulating parallel programming applications.

These applications include:

- CPU-bound process – This program computes `sin(x)` recursively. When run, the CPU utilisation is about 100%.

- Network-bound process – In the cluster, home directories are located on a NFS file server. The main network traffic is that generated by file operations. We developed a program that has low CPU utilisation and high network traffic. It reads a large file using the `fread()` function in the C library. In this program, the buffer is set to a large size. When reading the large file, it will cause high network traffic but moderate CPU use because the CPU has to wait on the transfer of data over the network. The average CPU utilisation is about 50%. The file transfer rate is about 10,000 Kbps.

  We also created a script that copies two large files at the same time from one node to another node. The Linux `cp` command is used. The average CPU utilisation is about 10%, and the network transfer is about 2000 Kbps.

- Memory-bound process – The programs for simulating a memory-bound process uses the `malloc()` function to allocate 200 Mbytes or 300 Mbytes of memory. Then the processes gradually load the memory. Since the nodes only have 192 Mbytes of physical memory, the memory is exhausted, and virtual memory software has to move pages to swap space on a local disk.

- Mixed processes – One program recursively reads a small file every second. This simulates a process with low CPU utilisation and low network traffic. The average CPU utilisation is about 3%, and the average network traffic is about 300 Kbps.

  Another program simulates a program with high CPU utilisation and low network traffic. It reads a large file using `fread()`, but the buffer is set to a small size. The average CPU utilisation is about 100%, and the network traffic is about 300 Kbps.

The terms low and high with respect to utilisation are relative. That is, there is no absolute value that is considered low or high. Thus a CPU utilisation of 50% is higher than 10% but lower than 90%.

The tests consist of two parts:

- Workload – The workload of the tests includes a batch of programs which simulates a user's work. These programs are chosen from the above pool of programs as needed. The programs are randomly chosen each time. Between two programs, there is a random several seconds sleep time to simulate a user's thinking time.

- Background programs – A series of background programs are used to simulate different loadings of the nodes, such as some nodes with low CPU utilisation and low network traffic, some nodes with low CPU utilisation and high network traffic, some nodes with high CPU utilisation, and some nodes with high memory utilisation.

  The background programs and random workload processes can make a node's loading random. In this situation, the proposed load balancing algorithm can make each process in the workload choose the proper node from the different loadings of the nodes.

5.1 Detection of idle nodes

As previously discussed, CPU queue length might not reflect correctly whether a node is idle. For some processes, a node may have a low CPU utilisation but have high network traffic or high memory utilisation. For these kinds of processes, the process is normally not on the CPU queue when collecting information every second. The CPU queue length based load estimation policy will determine the nodes running these kinds of processes as idle nodes, although the node has high network traffic or high memory utilisation.

In addition, the type of each new process is unknown. Ideally if there are idle nodes, the new process should run on an idle node. If an idle node cannot be detected correctly, the performance will be degraded. For example, if a new process needs high memory size, and a node with high memory utilisation but zero CPU queue length is chosen as an idle node, the performance will be greatly reduced.

The methodology used by the proposed algorithm is to run a workload on the nodes one by one. These nodes have different loading characteristics. The purpose of this test is to see whether an incoming process can be allocated to an idle node correctly. The run time of the workload will be measured and used as the measurement of performance.

- Background programs – The background programs include three types of processes. They make nodes have low CPU utilisation and different network traffic and memory utilisation, but the CPU queue lengths are all zero. The loading of these nodes are:

  - Node27 is left idle. On this node, CPU utilisation is zero and CPU queue length is zero.
  - Node28 runs a program making it have low CPU utilisation, low network traffic, and a CPU queue length of zero.

Table 1. Run time for detection of idle nodes experiment

Model	node27	node28	node31	node33
Proposed	70.24	70.79	74.47	71.24
Queue-length based	70.65	74.33	418.09	79.14

  - Node31 runs a program making it have high memory utilisation and forcing memory paging to the hard disk. The CPU utilisation is low and CPU queue length is zero.
  - Node33 runs a program making it have low CPU utilisation, high network traffic, and CPU queue length is zero.

  Thus there is a group of four machines with different loading characteristics. However each node has a zero length CPU queue.

- Workload – A 20-program workload is randomly chosen from the pool of workload processes with different random seeds. The programs include different types of applications.

The 20-program workload is run on each of the nodes, one at a time. In other words, the 20-program workload is run on node27, leaving it to choose a target node according to the load balancing algorithm. Then the 20-program workload is run on node28, and so on.

The results are shown in Table 1. The result is the average run time of the workload being presented to each of the nodes.

For the proposed load balancing algorithm, different nodes may be chosen as target nodes for each new process. In fact, for this algorithm, node27 is the only "idle" node, and it is always chosen. When using node28, there is a small overhead associated with remotely starting the new processes on node27. When using node31, there is a slightly larger overhead waiting on the virtual memory system to allocate pages so the new processes can be remotely started on node27. For node33, there is a bit of waiting for the new processes to be remotely started on node27 over a busy network.

On the other hand, the CPU queue length based model always chooses the local node to run since the CPU queue length is zero. When using this model, node27 is truly idle and gives the best overall time. Node28 is lightly loaded and takes a longer time due to its background processes. Node31 takes a very long time because the virtual memory system is forced into paging to continue running the background processes and the new process. Node33 has some CPU overhead associated with network processing and takes longer than a truly idle node.

Thus the proposed load balancing algorithm performs better than a CPU queue length based algorithm in detecting truly idle nodes.

5.2 Different types of processes: CPU-Bound, Network-Bound and Memory-Bound

In this section, we assume the types of new processes are known. They are CPU-bound, network-bound, and memory-bound. The test is to explore the effect of the load estimation policy on different types of applications. We will compare the performance of the proposed model and the CPU queue-length based model.

The methodology is to run a similar set of programs on various nodes with different loading characteristics and compare the performance of the two load balancing algorithms.

- Background processes – The background processes are similar to the previous test. The difference is the addition of node29. The characteristics of the nodes are:
  - Node27 is left idle.
  - Node28 runs a program making it have low CPU utilisation and low network traffic.
  - Node29 runs a program making it have high CPU utilisation and low network traffic.
  - Node31 runs a program making it have high memory utilisation and forcing memory paging to the hard disk.
  - Node33 runs a program making it have low CPU utilisation and high network traffic.

- Workload - There are three groups of workloads:
  - CPU-bound workload – This workload consists of 10 arithmetic computing processes.
  - Network-bound workload – This workload consists of 10 processes using NFS to generate a high amount of network traffic.
  - Memory-bound workload – This workload consists of 10 processes which need a large amount of memory.

Each workload group runs its ten programs one at a time. This means that one process is started. The node waits for that process to complete, either locally or remotely. Then it starts the next process after a random wait. This cycle continues until all ten programs are run.

Table 2. Average run time for the test about different types of applications

Model	CPU-bound	Network-bound	Memory-bound
Proposed	25.52	61.63	235.58
Queue-length based	23.05	96.09	499.71
Precent change between models	-10.7	35.9	52.9

Each of the three groups of workloads are run separately. For each group (CPU, network, or memory bound), the tests are started approximately one second apart on all 5 nodes. After the start, there is no coordination between nodes in starting the remaining nine processes since there is a random wait between each new process on a given node.

Each of the tests are run twice and the average run time of each test is calculated and taken as the result. The three tests are:

- Test 1– Run a series of CPU-bound processes on all nodes at the same time.
- Test 2 – Run a series of Network-bound processes on all nodes at the same time.
- Test 3 – Run a series of memory-bound processes on all nodes at the same time.

The results are shown in the Table 2. It gives the average run times and the percentage change between the two load estimation policies. The results show that:

- When the incoming workload contains only CPU-bound processes, the performance of the proposed model is about 10.7% worse than the CPU queue-length based model. The reason for this is that CPU-bound processes mainly consume CPU time. The best node should be the node with the lowest CPU utilisation regardless of whether there is a high network traffic or high memory utilisation. The CPU queue-length based algorithm fits this workload type easily. That is, it will treat all nodes with low CPU utilisation and zero CPU queue length as idle nodes. While the proposed model tries to choose the node with low CPU utilisation and low network traffic. In addition, the node with a high memory utilisation is considered a highly loaded node. Thus the number of selectable nodes is lower and more processes have to run locally.
- When the workload is only a network-bound workload, the performance of the proposed algorithm is about 35.9% better than CPU queue-length based algorithm. That is because the proposed model will first choose the nodes with low CPU utilisation and low network traffic, while the CPU queue-length based model might choose a node with high memory utilisation or high network traffic. The network will become bottleneck if a network-bound process runs on a node with high network traffic. The performance will be reduced.
- When the workload includes all memory-bound processes, the performance of the proposed algorithm is about 52.9% better than the CPU queue-length based algorithm. The reason is that the CPU queue-length model will decide a node with zero CPU queue length but high memory utilisation as an idle node.

  When the physical memory of a node is exhausted, it needs to page. In this situation, a new memory-bound process running on this node makes the performance significantly worse.

We can conclude that if all applications of the cluster are CPU-bound processes, CPU queue-length based model may be a good choice. Otherwise the proposed algorithm shows superior performance. In general, a real cluster may have network traffic and some nodes may be using a significant amount of memory. Therefore network traffic and memory utilisation are worth considering in a load balancing algorithm.

5.3 Mixed types of applications

In this section, we explore the effect of the load estimation policy on mixed types of applications. We will compare the performance of the proposed algorithm and the CPU queue-length based algorithm with a mix of application types. The applications include arithmetic computing, programs using network filesystem, and programs using a large amount of memory.

The methodology is the same as Section 5.2. The difference is that the workload on each node is mixed. The workload consists of 10 programs. Each program in the workload is randomly selected from the pool of programs. Thus we do not know the type of each new process a priori. The type of program is decided at run time. The workload runs on the nodes which have the same loading characteristics as in Section 5.2. Each test is run four times with a different random seed each time.

The result is shown in Table 3. Each node has two values, proposed algorithm and CPU queue-length based algorithm. They are the average of four runs. The `Total time` row is the total run time of the five nodes above. The `Diff time` row is the maximum difference between the quickest and slowest nodes.

Table 3. The test about mixed applications

Items	Proposed model	Queue-length based model
node27	922.9	878.37
node28	1120.43	1048.77
node29	1553.09	3129.31
node31	1387.26	4391.13
node33	1626.52	2258.53
Total time	6610.2	11706.11
Diff time	703.62	3512.76

The cluster performance with the different algorithms can be shown by the total run time of all programs on all cluster nodes. The maximum differential of the run time of each node can indicate the balance of the load on each node.

According to the tests, the result shows that the overall performance of the proposed algorithm is about 43.5% better than the CPU queue-length based algorithm. The maximum differential of the run time of proposed algorithm is far better than the CPU queue-length based algorithm. This indicates the proposed algorithm more effectively uses the cluster than the CPU queue-length based algorithm. Only when all three of CPU utilisation, network traffic and memory utilisation are low is the CPU queue-length algorithm better. When there is a reasonable amount of network traffic and/or memory utilisation, the proposed algorithm has better performance.

6 Conclusion

This research has developed an efficient load balancing system. For this system, we developed a new way to calculate the load of cluster nodes. It uses CPU utilisation, CPU queue length, network traffic, and memory utilisation to determine the load of each node instead of traditional CPU queue length.

A number of tests were performed. From these tests, we can conclude:

- Effectively using the idle nodes in the cluster is important. To determine the idle nodes, CPU utilisation is usually better than CPU queue length. CPU utilisation can reflect a node with low CPU use.
- For a cluster environment, if the main applications include the CPU-bound process, network-bound and memory-bound process, and the new processes are unknown, the load balancing system based on CPU queue length, CPU utilisation, network traffic, and memory utilisation performs better than CPU queue length.
- For a cluster which is mainly used by CPU-bound processes such as science computing, the load balancing system based on CPU queue length performs better. Otherwise the proposed load balancing system performs better.
- When deciding a transfer process, it is efficient to let highly loaded nodes transfer their processes to other lightly loaded nodes. This allows highly loaded nodes to have more chances to remove their local load.

References

[1] M. V. Devarakonda and R. K. Iyer. Predictability of processs resource usage: A measurement-based study on UNIX. *IEEE Transactions on Software Engineering*, 15(12):1579–1586, 1989.

[2] D. Ferrari and S. Zhou. An empirical investigation of load indices for load balancing applications. In B. A. Shirazi, A. R. Hurson, and K. M. Kavi, editors, *Scheduling and Load Balancing in Parallel and Distributed Systems*, pages 487–496. IEEE Computer Society Press, Los Alamitos, California, 1995.

[3] O. Kremien, J. Kramer, and J. Magee. Scalable, adaptive load sharing for distributed systems. *IEEE Parallel and Distributed Technology, Systems and Applications*, 1(3):62–70, 1993.

[4] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, 1991.

[5] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, California, 1995.

[6] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[7] P. Werstein, M. Pethick, and Z. Huang. Locabus: A kernel to kernel communication channel for cluster computing. In *Proceedings of the Fifth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 497–504, Singapore, 2004.