J Supercomput (2011) 56: 270–299
DOI 10.1007/s11227-009-0369-3

On construction of a well-balanced allocation strategy for heterogeneous multi-cluster computing environments

Chao-Tung Yang · Kuan-Chou Lai · Hao-Yu Tung

Published online: 18 December 2009

Abstract With the rapid increment of the heterogeneity of hardware devices, cluster computing has to encounter the problem of handling heterogeneous resources for exploiting the utilization of system resources. This paper introduces a new job allocation strategy based on multi-clusters in diskless environments. By adopting Ganglia as the resource monitor and Condor as the queue system, a heterogeneous multi-cluster system is also constructed with and without storage devices for evaluating the system performance. The proposed algorithm is called the Well-Balanced Allocation Strategy (WBAS) in which the scheduler dispatches MPI-based jobs to appropriate resources across multi-clusters. The strategy focuses on dispatching jobs to nodes with similar performance, thus equalizing execution times among all the required nodes. The WBAS is implemented on the constructed heterogeneous multi-cluster system to evaluate the performance of the scheduling strategy. The experimental results show that the proposed strategy performs well and could efficiently improve the system performance.

Keywords Multi-cluster · Heterogeneous · Job scheduling · Resource monitoring

C.-T. Yang (✉) · H.-Y. Tung
High-Performance Computing Laboratory, Department of Computer Science, Tunghai University, Taichung 40704, Taiwan (ROC)
e-mail: ctyang@thu.edu.tw

H.-Y. Tung
e-mail: jamesmap.tw@gmail.com

K.-C. Lai
Department of Computer and Information Science, National Taichung University, Taichung 40306, Taiwan (ROC)
e-mail: kclai@ntcu.edu.tw

1 Introduction

Cluster computing in commercial and non-commercial applications becomes more and more common these days [2–5, 10, 13, 16, 18–21]. Multi-cluster systems integrate multiple clusters into a huge environment for research facilities to make use of high-performance computing resources by matching the computational requirement. As computer architectures become more and more diverse and heterogenic, and computer expiration rates are higher than before, the old and unused computers could be used for increasing system computing capability. Therefore, clusters usually consist of computers with different processors, memories and hard disk drives. It causes difficulty for dealing with such heterogeneity in a cluster.

As is known, heterogeneous clusters have differing computing power and characteristics. MPI-based job is usually split into several tasks to be executed with the same program file in parallel. Since the heterogeneous nodes have differing computing capacities, the time to execute the same job in different nodes is different. As a result, faster nodes of cluster have to wait for slower nodes, thus wasting the computing resources [1, 6, 7, 11, 12, 14, 15, 17].

This paper addresses the resource wasting problem. The proposed scheduling strategy first gathers all needed information from each node, depending on the scheduler. Depending on this gathered information, the proposed strategy decides which jobs could be dispatched to which machines. The Well-Balanced Allocation Strategy (WBAS) tries to achieve the goal that jobs finish in the nearly same times, thus shortening the idle times of faster nodes.

This paper introduces a job allocation strategy, named the Well-Balanced Allocation Strategy (WBAS), to increase the efficiency for the multi-cluster environment. The proposed strategy gathers the information needed from all the execution nodes including the CPU, free memory, loading, and network status. According to node's computing power, WBAS classifies the nodes of clusters in the multi-cluster system into different levels. And then, the scheduler dispatches the job based on three policies adopted in WBAS. The decision is made based on whether there are sufficient free nodes in the clusters or not; the co-allocation is made to submit the job across multiple clusters. Finally, the proposed strategy is evaluated in a multi-cluster environment for the feasibility of this job allocation strategy.

The rest of this article is organized as follows. In Sect. 2, we mention some technologies and our cluster layer in brief. Section 3 describes the system architecture and the scheduling algorithm. The experimental results are shown in Sect. 4. In Sect. 5, the conclusions and the future work are given.

2 Background review

2.1 Cluster computing

Cluster computing supports high-performance parallel computing based on inexpensive computer hardware. A Beowulf cluster system includes a group of usually identical PC computers which run the same Free and Open Source Software (FOSS)

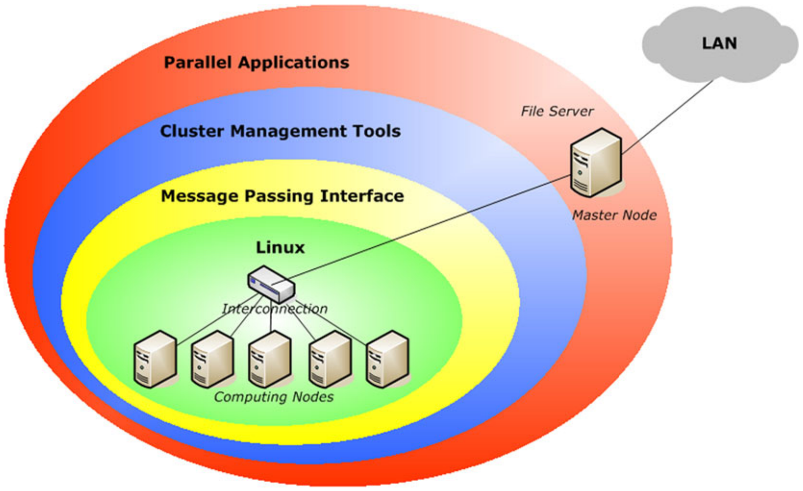


Fig. 1 Logic view of a PC cluster

Unix-like operating system, such as BSD, Linux or Solaris. They are networked by a small TCP/IP LAN, and have the same libraries and programs installed which allow processes to be shared among them.

A Beowulf cluster uses a multi-computer architecture, as depicted in Fig. 1. It consists of one or more head nodes and available tail nodes, compute nodes, or cluster nodes, which are interconnected via widely available network. All the nodes in a typical Beowulf cluster are commodity systems-PCs, workstations, or servers running commodity software such as Linux, as mentioned above [3, 4, 10, 16, 19].

The head node acts as a server for NFS and as a gateway to communicate with the outside world. As an NFS server, the head node provides the user file space and other common system software for the computing nodes via NFS. As a gateway, the head node allows users to gain accesses through it to the computing (tail) nodes. Usually, the head node is the only machine with a second network interface card (NIC) to connect to the outside world. The sole task of the tail nodes is to execute parallel jobs.

From a user's perspective, a Beowulf cluster appears as a Massively Parallel Processor (MPP) system [2]. The most common methods of using such a system are to access the head node either directly or through Telnet or remote login from personal workstations. Users can prepare and compile their parallel applications, and also spawn jobs on a desired number of tail nodes in the cluster. Applications must be written in parallel form by the message-passing programming model. Jobs in a parallel application are spawned on compute nodes, which work collaboratively until finishing the application. During the execution, computing nodes use the standard message-passing middleware, such as Message Passing Interface (MPI) [28, 32] and Parallel Virtual Machine (PVM) [34], to exchange information.

Commonly used parallel processing libraries include MPI (Message Passing Interface) and PVM (Parallel Virtual Machine). Both of these libraries permit the programmer to divide a task among a group of networked computers, and recollect the processing results. Different libraries based on the MPI standards exist, for example, MPICH [33] and LAM/MPI [31]. In general, there is a misconception that arbitrary software runs faster on a Beowulf system. Software must be revised to take advantage of the cluster. Specifically, it must perform multiple independent parallel operations that can be distributed among the available processors.

2.1.1 Message passing interface

There are two main Message Passing Interfaces used for executing MPI jobs under Linux, MPICH and LAM. They are briefly described as follows. MPICH is a robust and flexible implementation of the MPI. MPI is often used with parallel or distributed computing projects. MPICH is a multi-platform, configurable system (development, execution, libraries, etc.) for MPI. It can achieve parallelism using networked machines or using multitasking on a single machine. LAM is an implementation of the Message Passing Interface parallel standard that is especially friendly to clusters. It includes a persistent runtime environment for parallel programs, support for all of MPI-1, and a good chunk of MPI-2, such as the dynamic functions, one-way communication, C++ bindings, and MPI-IO.

2.1.2 DRBL

DRBL stands for Diskless Remote Boot with Linux [23, 29]. This solution is solely designed and implemented by people in National Center of High-performance Computing (NCHC), Taiwan. DRBL uses PXE or Etherboot, NFS, and NIS to provide services for client machines [26]. When the server is a DRBL server, the client machines can boot a Linux image over the network via PXE or Etherboot (Diskless). DRBL does not touch the hard drive of the clients, so other OS (for example, MS Windows) installed on the client machines will retain their own OS, or clients can do without hard drives entirely. This may be important in a phased deployment of the GNU/Linux, where users still want to have the choice of booting as an MS Windows system and running the Office application. DRBL allows to be flexible in one's deployment of the GNU/Linux system.

2.1.3 Ganglia

The Ganglia project grew out of the University of California, Berkeley's Millennium initiative [30]. Ganglia is a scalable open source distributed system for monitoring the status of nodes (processor collections) in wide-area systems based on clusters. It adopts a hierarchical tree-like communication structure among its components in order to accommodate information from large arbitrary collections of multiple clusters, such as Grids. The information collected by the Ganglia includes hardware and system information, such as processor type, load, memory usage, disk usage, operating system information, and other static/dynamic scheduler-specific information.

2.1.4 Condor

Condor [12, 14, 15, 20] is an Open Source project developed by the Condor team of Computer Sciences Department at the University of Wisconsin-Madison, WI. Condor is a powerful workload management system for computing intensive jobs. It is useful for constructing a multi-cluster environment; it also provides many features for handling the nodes in cluster pools, such as queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management.

Condor provides a job queue in our system. Under the assumption that only one job is allowed to execute one node, ordinary way of submitting jobs under the cluster could only be done with one user at a time. But by using Condor, we are able to submit up to one hundred jobs at a time. Condor automatically queues the jobs in its queue waiting for the jobs which were submitted prior to the finish. Moreover, the jobs can also be monitored through Condor, and also the removing of the jobs.

2.2 Related works

The personal computer becomes cheaper today, and the ratio of cost to price is even more valuable when we use the Beowulf cluster for parallel processing and applications. By purchasing a number of PCs to build a Beowulf cluster in batches, we could not obtain higher performance because we cannot easily integrate these heterogeneous computational resources. We have several ways to integrate the heterogeneous clusters to build a new computational resource to get a better performance [13]. It is a problem to build the Multi-Cluster Computing. The purpose of building the Multi-Cluster Computing is to get any type of resources to resolve many of demanding tasks [8, 9].

The major objective of scheduling in a multi-cluster environment is to make the best use of resources [24, 25]. When a job arrives at the job queue, the responsibility of the scheduler is to decide which machine should handle it [27]. Many studies have been conducted on multi-cluster scheduling. Generally, they fall into one of the two categories: static scheduling or dynamic scheduling [5]. Static scheduling, performed at compiling time, allocates jobs to individual processors before execution; and the allocation remains unchanged during execution. Dynamic scheduling is performed at runtime and can support dynamic load balancing and fault tolerance. Although load balancing could distribute the load to different nodes during execution [18, 22], it introduces additional overhead into program execution. The method we propose here is based on static scheduling.

Most static scheduling methods must gather information for the scheduler to make decisions on what kinds of resources are necessary. Some of them use memory latency as the main indicator for load sharing [17], or take memory and CPU power usage into account [6]. Others take further steps like queuing [21] or co-allocation [11] into consideration. In single-cluster systems there is no need to consider co-allocation, which allocates jobs across clusters, but it is necessary in multi-cluster systems. Although the bandwidth could be disregarded in multi-cluster systems, it does have some effect on systems with network-bounded jobs.

Certain studies have focused on queuing systems. Some of them consider jobs in the waiting queue [1]. Most multi-cluster environments are scaled hierarchically, working by methods for placing and maintaining queues [7]. They also divide jobs into two categories: single and multiple jobs, which is practical for real-world environments. Many scheduling policies have been proposed. The main goal is to make the best use of the resources, and that is also the major purpose of the present work.

3 Well-balanced allocation strategy

3.1 System architecture

Our system performs two services: a monitoring service and a job managing. The user could acquire node statuses before submitting jobs to the system. As shown in Fig. 2, at any time the user can browse the status of the whole cluster via the monitoring service. The job manager checks the status of jobs in the job queue. The job queue handles all submitted jobs, including job status monitoring and job removing. The scheduler uses a simple FCFS strategy and automatically chooses nodes or clusters for executing the first job in the queue [35–37].

Our system consists of four clusters, each with one head node and seven tail nodes. The clusters are named “amd1,” “amd-mpdual1,” “condor1,” and “condor2.” All have AMD ATHLON processors except the two head nodes of “condor1” and “condor2,” which use both Intel Pentium CPUs. The “condor1” and “condor2” clusters have hard disks only on their head nodes; the seven tail nodes have none. Thus, these two clusters need a diskless environment in order to work. There is one master node on top

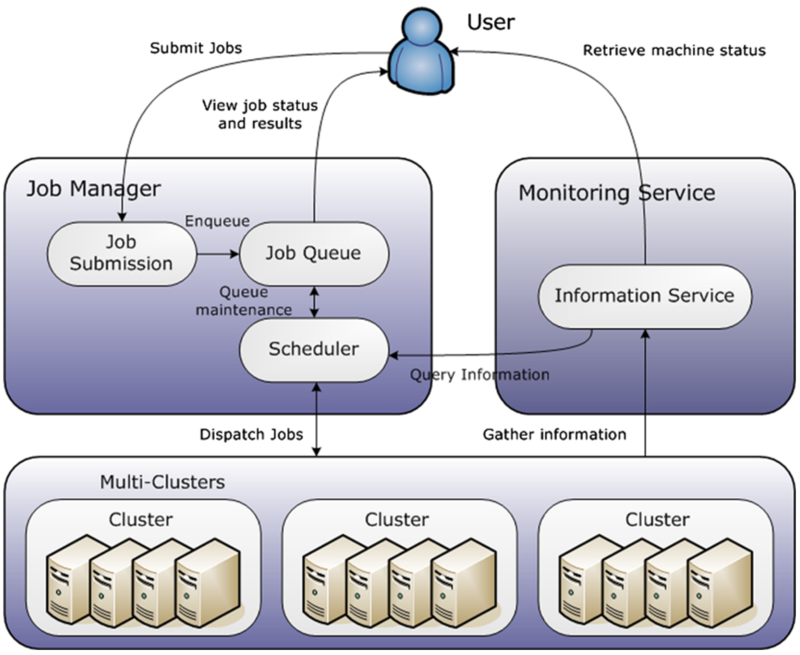


Fig. 2 System architecture

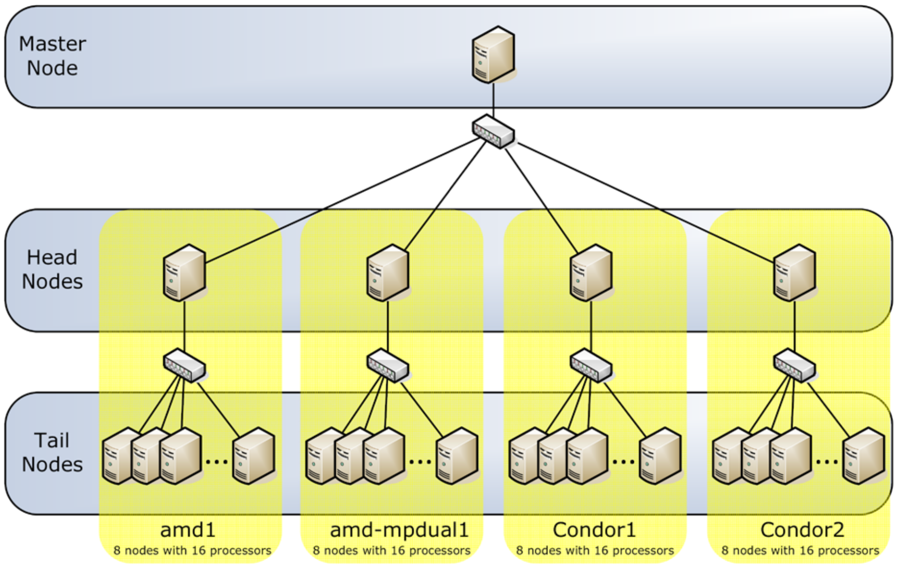

Master Node
Head Nodes
Tail Nodes
amd1
8 nodes with 16 processors
amd-mpdual1
8 nodes with 16 processors
Condor1
8 nodes with 16 processors
Condor2
8 nodes with 16 processors

Fig. 3 Three layers of multi-cluster view

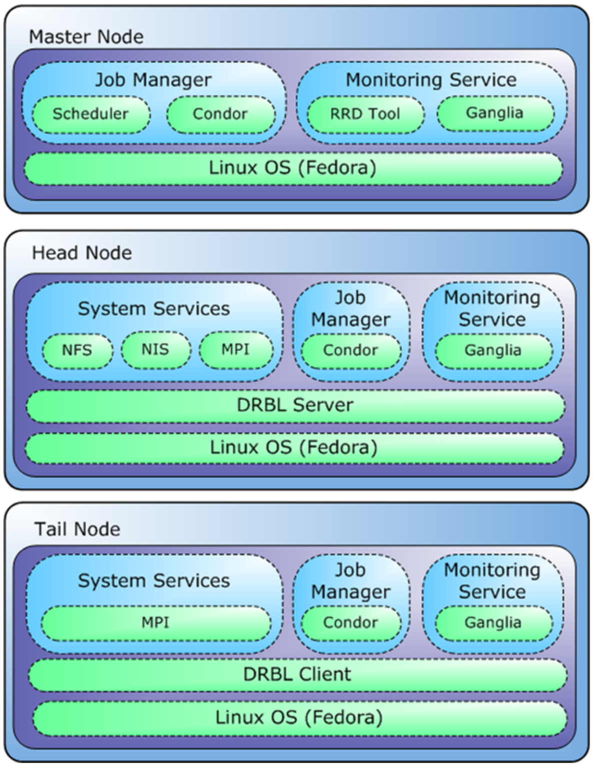
of all clusters that handles job assignments. All nodes are linked by a gigabit network, as shown in Fig. 3. Since we have few nodes, they are basically placed hierarchically.

The Linux system is installed on all nodes. The head nodes are responsible for providing services to their clusters, and the tail nodes are the basic computational nodes. The master node is responsible for deploying jobs and offering monitoring services. As shown in Fig. 3, the clusters are composed of three layers: one master node, head nodes, and tail nodes.

As described above, the software stack diagram is shown in Fig. 4. We divided the three layers as following:

- Master Nodes: The master nodes support the main system functions, the monitoring service and the job manager. They include the Ganglia monitoring server and the Web page with a RRD tool. Condor is also deployed on these nodes.
- Head Nodes: They provide the needed services for the clusters, including the Network Information Service (NIS), Network File System (NFS), and the Message Passing Interface (MPI) environment needed to run MPI jobs. Ganglia and Condor are also implemented to keep users informed on node statuses and to schedule jobs on the clusters. DRBL enables "condor1" and "condor2" to run as diskless nodes, i.e., the execution of Linux without a hard disk. In our situation, only one hard disk is required per cluster.
- Tail Nodes: The computational nodes on the bottom layer are responsible for executing jobs and sending their node statuses. The working file systems on these nodes are mounted on NFS servers provided by the head nodes. Ganglia client-side services and Condor execution services are the only functions implemented on these nodes. MPI is deployed on these nodes to enable them to run MPI jobs.

Fig. 4 Software stack



The main goal of scheduling systems is to make the best use of resources. The main functions of a scheduler include resource discovery, information gathering, system selection, and job execution. Information is gathered for the scheduler to make the best decision in determining which nodes' jobs are assigned. Therefore, this study proposes a scheduling policy called the Well-Balanced Allocation Strategy (WBAS). Choosing the most efficient nodes to execute jobs is a simple method for single jobs, but it does not always work well in MPI environments because MPI jobs are executed on multiple/single nodes; so choosing the most powerful nodes is not always practical. It may be feasible for homogeneous and dedicated clusters, since all their nodes are identical. But in heterogeneous environments, selecting nodes with similar performance is a more complex work. Our objective is to even out the computation times of all executing nodes. The main problem of MPI jobs is that when executed in heterogeneous or non-dedicated environments, nodes will have differing computing capacities. Therefore, the distributed loads may be different. Some nodes would finish their jobs earlier waiting for slower nodes.

The major objective of our policy is to find nodes with similar performance to execute jobs for improving the resource utilization. When a job in a queue is ready for deployment, we first gather the information needed for decision-making. The scheduler then fetches the processor request required for the job and the scheduling policy starts to decide which nodes accept the job for execution. We use the gathered information to calculate the performance value for every node. Then we classify all these

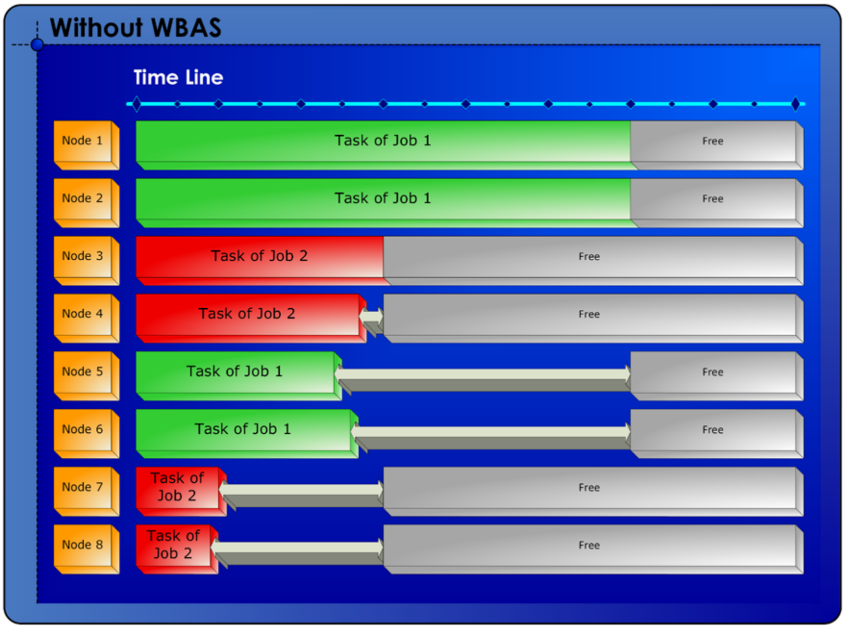


Fig. 5 An example without WBAS

values into α levels, from the fastest to the slowest. The scheduler decides which policy to use in deploying the job to a specific node.

Our strategy aims to improve the resource utilization and reduce the job turn-around time in heterogeneous multi-cluster systems. Clusters with heterogeneous nodes or non-dedicated clusters have the most effectiveness in our scheduling strategy. The chief weakness of running MPI programs is in that when employing nodes with different computing power or loading, the nodes that are faster or with lighter loads are more likely to finish earlier than the nodes that are slower or with heavier loads. This will cause the nodes finishing earlier to wait for the nodes that have not finished their job. Figure 6 shows an example of the scheduling without WBAS.

Assume that we have eight nodes with different computing power. Nodes 1 to 4 are the slower nodes, or in other words, machines with weaker computing power. On the other hand, nodes 5 to 8, are the nodes with higher computing power. In Fig. 6, two jobs are submitted to these eight nodes. These two jobs both requested four nodes for execution. The timeline shows how long the jobs need to run under certain nodes. We can see that if the nodes which these two jobs acquired are of a different computing power, then nodes 5 and 6 have to wait for nodes 1 and 2, nodes 7 and 8 have to wait for nodes 3 and 4. The double arrowhead line shows the time we have lost during this execution procedure. And that is also the main problem that we are trying to solve in this paper.

As shown in Fig. 7, the time we have lost is nearly zero in the same example. We have earned a lot of free execution time from nodes 5 to 8 for other users to submit their jobs to these nodes. Nodes 1 to 4 have similar computing powers, as well as

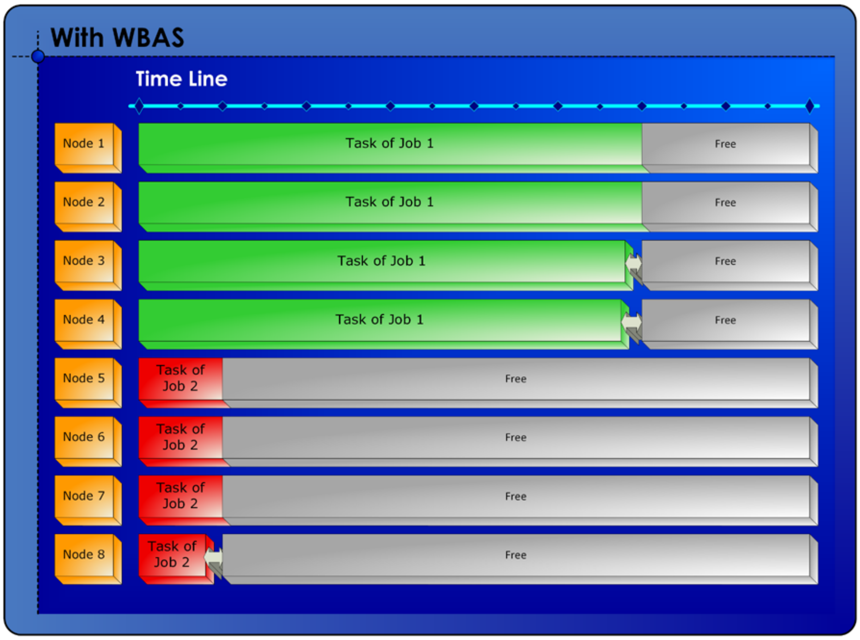


Fig. 6 An example with WBAS

nodes 5 to 8. WBAS will calculate the performance and the loading of all the free nodes. And it will submit the jobs to suitable nodes with the gathered information.

3.2 Parameters and the algorithm

Now, we discuss the evaluating process of the performance power of all the nodes. The parameters used in the algorithm are listed as follows:

- job_i: The ith job is de-queued from the queue, where $i = 1 \sim n$. The job contains some information such as the job name, program name, program arguments, input data, and number of required processors. The program is usually a parallel program written by the MPI library and compiled by MPICH or LAM/MPI. The scheduler will allocate resources according to the information provided by the job.
- NP_{req}: The number of nodes required to execute job_i. When the scheduler successfully dispatches job_i, the distributed resources used for job_i will be locked until job_i is finished. In other words, we allow only rigid job scheduling by pure space sharing, which means that one processor is permitted to run one job at a time.
- NP_{all}: The total number of processors in the multi-clusters. If NP_{req} exceeds NP_{All}, the job is dropped.
- $SN_{\max}$: The maximum number of nodes in a single cluster on the same level.
- $PN_{\max}$: The maximum number of nodes in a cluster pair on the same level.
- $MN_{\max}$: The maximum number of nodes in all of cluster pairs on the same level.
- $Load_i$: The sum of $load^1$, $load^5$, and $load^{15}$ on $Node_i$, where $Node_i$ is the ith node and is available for allocation; calculation of $Load^i$ is as follows: (here $load^1$ is the

ith processor loading in one minute, $load^5$ is the ith node loading in five minutes, and $load^{15}$ is the ith node loading in fifteen minutes).

$$Load_i = \frac{load_i^1 \times 15 + load_i^5 \times 5 + load_i^{15}}{21} \tag{1}$$

- CP_i: The computing capacity of the ith node; its calculation is as follows:

$$CP_i = \frac{HPL}{Cpu_{\text{num}}} \times \frac{100 - Load_i}{100} \times (Mem_{\text{free}} + Cpu_{\text{pow}}) \tag{2}$$

- *HPL*: The site benchmarking value obtained by the approach of benchmarking.
- Cpu_{num}: The total number of CPUs in a specified cluster. Job "the unit of job distributed" is a node.
- Mem_{free}: Available node memory in gigabytes; if a node has 1.2 GB of free memory, this value will be 1.2.
- Cpu_{pow}: Node CPU clock rate in gigahertz; if a node has a clock rate of 1.8 GHz, this value will be 1.8.
- *LS*: Since we sort α levels according to computing power, this value is the size of each level.

$$LS = \frac{(CP_{\max} - CP_{\min})}{\alpha} \tag{3}$$

- Level: it means familiar computing node in the same group.
- α: The number of levels into which we sort all the nodes; the value depends on the number of clusters and their homogeneity. The default value is the number of clusters in the environment, which is 4 in our case.
- $CP_{\min}$: The lowest *CP* value among all nodes.
- $CP_{\max}$: The highest *CP* value among all nodes.
- BW_{mn}: The average bandwidth between the head node of cluster m and cluster n at a given time period. The value k stands for the number of measurements computed to get the average bandwidth; the default value is 30. The bandwidth is constantly being measured; BW_{mn} between two clusters is the bandwidth between them over the last two minutes.

$$BW_{mn} = \sum_{t=1}^{k} \frac{L_{mn}[t]}{k} \tag{4}$$

- $L_{mn}[t]$: The bandwidth between cluster m and cluster n at time t.

We sort all nodes by α levels according to their performance capacity from high to low. The procedure below shows how nodes are classified into α levels. The level classification function is responsible for sorting. Figure 7 shows nodes sorted into groups according to the *CP* value.

After sorting out all the available nodes in our system, we know which node belongs to which level. In other words, we classify all the nodes into α levels. The primary purpose for sorting nodes into different levels is to equalize execution times. The α value will be different for different environments, depending on the computing power of the specific multi-cluster system. When building clusters, we generally

Fig. 7 Level classification algorithm

```
**LevelClassify**($CP_i$){
  **if** ($CP_i \leqq CP_{\min} + LS$) **then**
    $Level_1 = i$
  **elseif** ($CP_i \leqq CP_{\min} + LS \times 2$) **then**
      $Level_2 = i$
  **elseif** ($CP_i \leqq CP_{\min} + LS \times 3$) **then**
        $Level_3 = i$
        ...
  **else**
          $Level\ \alpha = i$
        ...
      **fi**
    **fi**
  **fi**
}
```

place nodes with similar computing capacities in the same cluster. Machines with the greatest capacity are grouped into one cluster, those with slightly lesser capacity into the next, and so on. In such cases it is feasible to simply set α to the number of clusters in the environment.

3.3 The flow chart

The WBAS allocation strategy includes three main phases: information gathering, level classification, and allocation policy. There are two allocation policies: Single Cluster and Cluster Pair. Figure 8 shows a simplified flow chart. Jobs are dispatched from the master node's queue. The master node is responsible for making dispatch decisions, including fetching states and all node information, and for determining whether the number of processors requested is greater than the number of available processors in the whole cluster. If not enough processors are allocated, the job is abandoned and the user is informed that not enough processors are available to execute the job. The scheduler then begins the Well-Balanced Allocation Strategy to calculate the performance capacities of all the nodes.

The "level classification" function gathers the needed information about classification by the *CP* value for each node. After sorting all the nodes according to the α level, one of two policies may be chosen for the job. The first is the "single site only," and the other is "co-allocation" which means selecting more than one cluster for execution.

"Single cluster policy" is to select nodes in a single cluster on the same level using the best-fit algorithm. If there are not enough free nodes to use this policy, the "cluster pair policy" will be chosen.

"Cluster pair policy" is similar to the "single cluster policy", except that nodes from two clusters are chosen instead of one from the same level. Cluster pairs that satisfy the number of requested nodes are first selected. The BW_{nm} values of all the candidates are compared and the cluster pair with the lowest latency is chosen. The job is then dispatched to the nodes for execution.

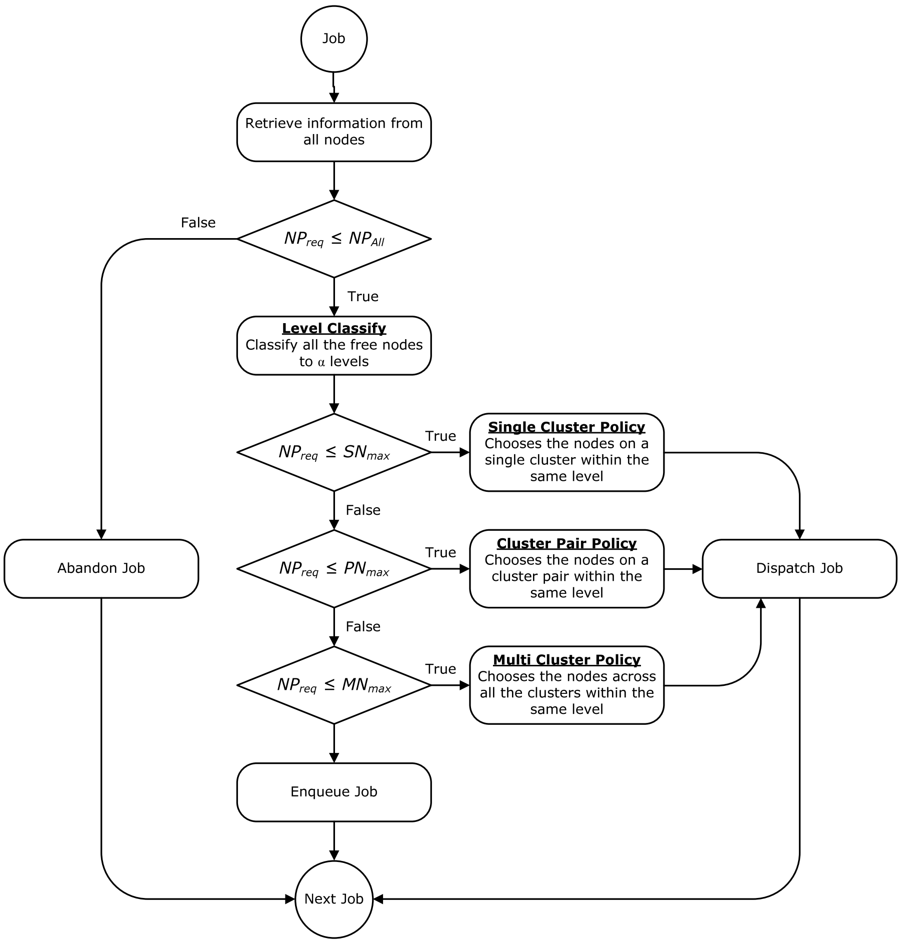

Job
Retrieve information from all nodes
$NP_{req} \le NP_{All}$
False
True
Level Classify
Classify all the free nodes to α levels
$NP_{req} \le SN_{max}$
True
Single Cluster Policy
Chooses the nodes on a single cluster within the same level
False
$NP_{req} \le PN_{max}$
True
Cluster Pair Policy
Chooses the nodes on a cluster pair within the same level
Abandon Job
Dispatch Job
False
$NP_{req} \le MN_{max}$
True
Multi Cluster Policy
Chooses the nodes across all the clusters within the same level
Enqueue Job
Next Job

Fig. 8 WBAS flow chart

"Multi-cluster policy" is to emphasize when users need a huge amount of nodes. This method tries to use all the clusters in the system but still within the same level. If neither policy met job requirements, the job is sent back to the queue to wait in line for another try.

3.4 An example

In this simple example we show how the WBAS works in real environments. Note that we have four clusters: Alpha, Bravo, Charlie, and Delta, each with eight nodes, for a combined total of 32 nodes in our multi-cluster environment. Figure 9 shows that the nodes are sorted into four levels, which means that α value is set to 4 in this case.

The scheduler is enabled whenever there is a job in the job queue, and checks for available nodes, which is, nodes not executing jobs. Some nodes in Fig. 9 are in the

Fig. 9 WBAS example

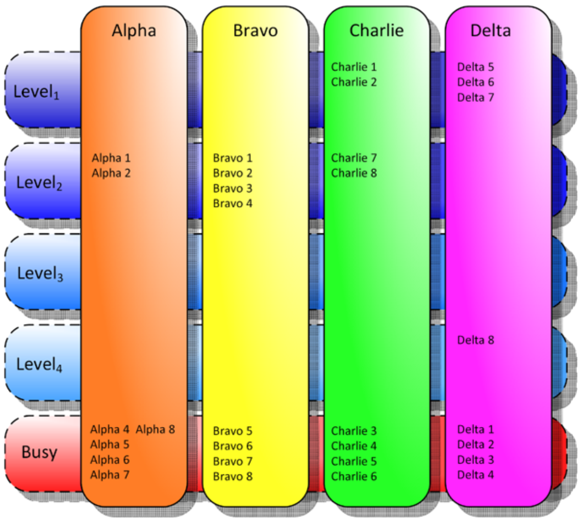


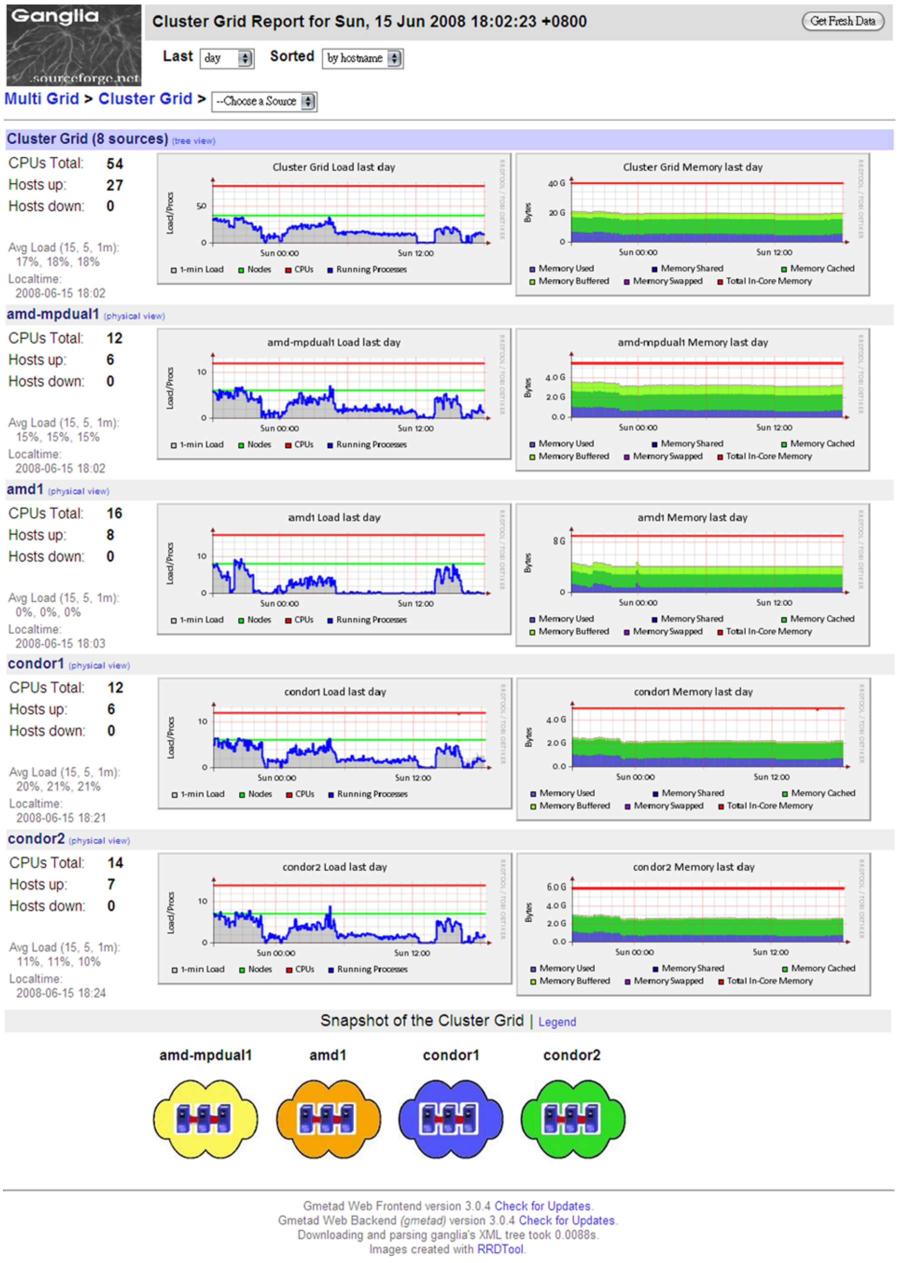
Table 1 The CP-value example

Node	*CP* value
Alpha 1	14.58
Alpha 2	15.64
Bravo 1	16.22
Bravo 2	15.15
Bravo 3	15.66
Bravo 4	16.71
Charlie 1	20.19 ($CP_{\max}$)
Charlie 2	19.88
Charlie 7	14.55
Charlie 8	15.27
Delta 5	19.01
Delta 6	18.97
Delta 7	18.55
Delta 8	8.11 ($CP_{\min}$)

busy state, which means they are running jobs. Therefore, the scheduler only gathers information on nodes Alpha 1∼2, Bravo 1∼4, Charlie 1∼2, Charlie 7∼8, and Delta 5∼8, which are free. Using the data gathered the scheduler then calculates *CP* values for these 13 nodes. As shown in Table 1, the highest ($CP_{\max}$) is for Charlie 1 at 20.19 and the lowest ($CP_{\min}$) for Delta 8 at 8.11.

After the *CP* values have been calculated, the nodes are sorted into four levels from high to low, as shown in Fig. 10. Below, we give examples for two NP_{req} values (number of user-requested nodes).

- Example 1. ($NP_{\text{req}} = 4$) If the enqueued job we are about to process needs four nodes for execution, Fig. 9 shows that we should choose Bravo 1∼4 based on the single-site policy of our WBAS.



Ganglia sourceforge.net

Cluster Grid Report for Sun, 15 Jun 2008 18:02:23 +0800 Get Fresh Data

Last day Sorted by hostname

Multi Grid > Cluster Grid > --Choose a Source

Cluster Grid (8 sources) (tree view)

CPUs Total: 54
Hosts up: 27
Hosts down: 0

Avg Load (15, 5, 1m):
17%, 18%, 18%
Localtime:
2008-06-15 18:02

amd-mpdual1 (physical view)

CPUs Total: 12
Hosts up: 6
Hosts down: 0

Avg Load (15, 5, 1m):
15%, 15%, 15%
Localtime:
2008-06-15 18:02

amd1 (physical view)

CPUs Total: 16
Hosts up: 8
Hosts down: 0

Avg Load (15, 5, 1m):
0%, 0%, 0%
Localtime:
2008-06-15 18:03

condor1 (physical view)

CPUs Total: 12
Hosts up: 6
Hosts down: 0

Avg Load (15, 5, 1m):
20%, 21%, 21%
Localtime:
2008-06-15 18:21

condor2 (physical view)

CPUs Total: 14
Hosts up: 7
Hosts down: 0

Avg Load (15, 5, 1m):
11%, 11%, 10%
Localtime:
2008-06-15 18:24

Snapshot of the Cluster Grid | Legend

Gmetad Web Frontend version 3.0.4 Check for Updates.
Gmetad Web Backend (gmetad) version 3.0.4 Check for Updates.
Downloading and parsing ganglia's XML tree took 0.0088s.
Images created with RRDTool.

Fig. 10 Ganglia web front-end multi-cluster view

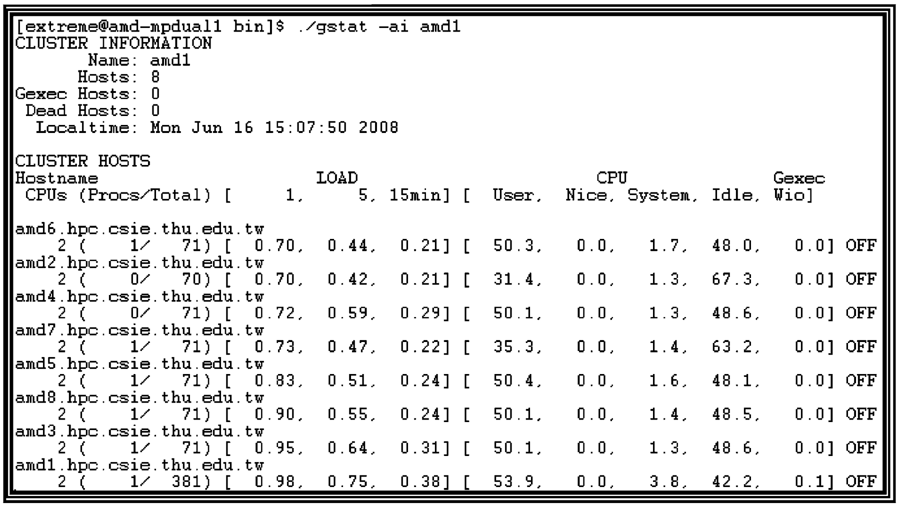
• Example 2. ($NP_{\text{req}} = 6$) In this case, the single-site policy will fail to satisfy the requirement since there are not enough free nodes on the same level in any of the clusters. The cluster pair policy of our WBAS must be used, which gives two choices, Bravo 1∼4 with Alpha 1∼2 or Bravo 1∼4 with Charlie 7∼8. From these, the pair with the higher *BW* value is selected. Since the bandwidth between Bravo and Charlie is high, they are chosen.

3.5 Monitoring and scheduler

Basically, we have two kinds of monitoring systems in our clusters. The first one is Ganglia which we briefly described in Sect. 2. And the second one is a simple monitoring system to fetch the remaining information which is needed by WBAS. Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It uses such information as the XML for data representation, XDR for compact, portable data transport, and RRDtool, to draw the graph on its web.

Figure 10 is shows all the clusters in each row. The first row displays the information of the whole multi-cluster system. There are 27 nodes with 54 CPUs in our multi-cluster. Each row represents the data of each cluster. We can see that there are up to eight clusters with different number of nodes in our system. The name of the cluster is displayed on each row and graphs. We can see there some "clouds" with the names of each cluster and the colors are different as well. These show the average load of each cluster, every color representing a different loading, e.g. orange for high loading and blue for light loading.

The other main purpose of using Ganglia is that we can fetch all the data we need from it. As mentioned before, Ganglia takes care of the data fetching throughout of the whole multi-cluster. The only thing we need to do is to deploy the system onto our multi-cluster platform. Ganglia provides a binary called "gstat," which can be used for our scheduling system to fetch the status needed. As in Fig. 11, this demonstrates an example of fetching the information of one of the clusters named "amd1." We have eight nodes in our cluster pool, numbered from one to eight. This information includes hostnames, CPU numbers, processes, loadings and CPU loadings.



```
[extreme@amd-mpdual1 bin]$ ./gstat -ai amd1
CLUSTER INFORMATION
       Name: amd1
      Hosts: 8
Gexec Hosts: 0
 Dead Hosts: 0
  Localtime: Mon Jun 16 15:07:50 2008

CLUSTER HOSTS
Hostname                     LOAD                       CPU              Gexec
 CPUs (Procs/Total) [     1,     5, 15min] [  User,  Nice, System, Idle, Wio]

amd6.hpc.csie.thu.edu.tw
    2 (    1/   71) [  0.70,  0.44,  0.21] [  50.3,   0.0,    1.7,  48.0,    0.0] OFF
amd2.hpc.csie.thu.edu.tw
    2 (    0/   70) [  0.70,  0.42,  0.21] [  31.4,   0.0,    1.3,  67.3,    0.0] OFF
amd4.hpc.csie.thu.edu.tw
    2 (    0/   71) [  0.72,  0.59,  0.29] [  50.1,   0.0,    1.3,  48.6,    0.0] OFF
amd7.hpc.csie.thu.edu.tw
    2 (    1/   71) [  0.73,  0.47,  0.22] [  35.3,   0.0,    1.4,  63.2,    0.0] OFF
amd5.hpc.csie.thu.edu.tw
    2 (    1/   71) [  0.83,  0.51,  0.24] [  50.4,   0.0,    1.6,  48.1,    0.0] OFF
amd8.hpc.csie.thu.edu.tw
    2 (    1/   71) [  0.90,  0.55,  0.24] [  50.1,   0.0,    1.4,  48.5,    0.0] OFF
amd3.hpc.csie.thu.edu.tw
    2 (    1/   71) [  0.95,  0.64,  0.31] [  50.1,   0.0,    1.3,  48.6,    0.0] OFF
amd1.hpc.csie.thu.edu.tw
    2 (    1/  381) [  0.98,  0.75,  0.38] [  53.9,   0.0,    3.8,  42.2,    0.1] OFF
```

Fig. 11 Gstat output of Ganglia

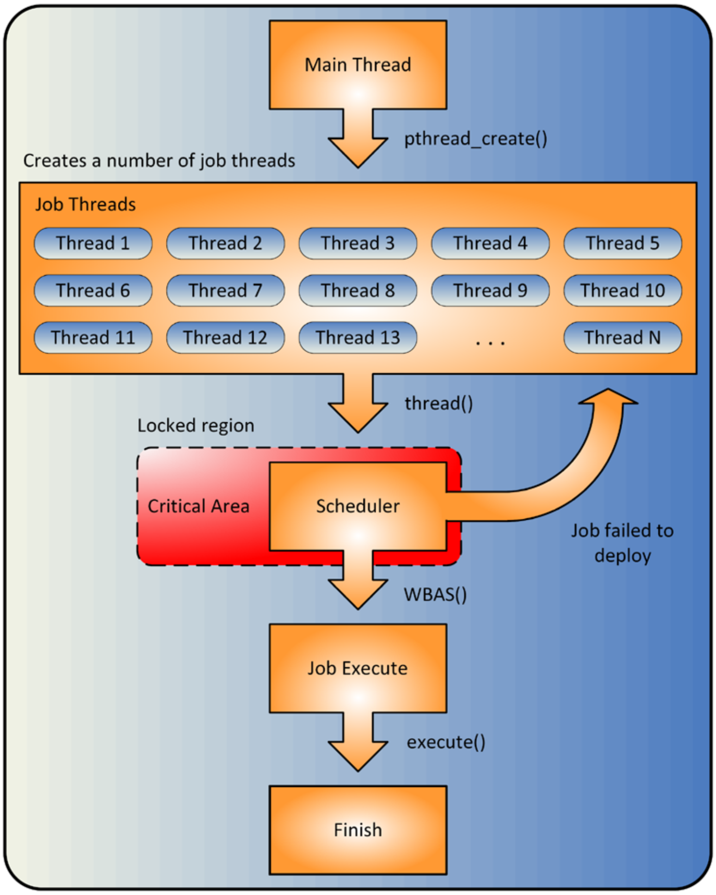


Fig. 12 The process flow of scheduler

Our scheduling system only needs to fetch the CPU number, one-minute loading, five-minute loading, and fifteen-minute loading. These data are then trimmed for our scheduler to use. These four values comprise the message we need to decide which nodes to deploy which jobs to our clusters. The master node fetches the information from gstat every time before making scheduling decisions.

The second kind of the monitoring system is used for our scheduling system. The primary objective of our scheduler is to fetch HPL and the memory-free value. Although NFS is operating on each cluster, the file systems are still separated from each cluster. So we use SSH (Secure Shell) to fetch the needed data. An easy shell script could do the job. The HPL value is stored on the head nodes, and the memory-free value is stored in a file "/proc/meminfo." The HPL value remains unchanged except when there are any hardware changes within this cluster. In this situation, we recommend to rerun the HPL benchmark program to get a new HPL value for this cluster. On the other hand, the file "/proc/meminfo" is changed within a second. This would not be a problem when we only need to fetch this data before deploying a job.

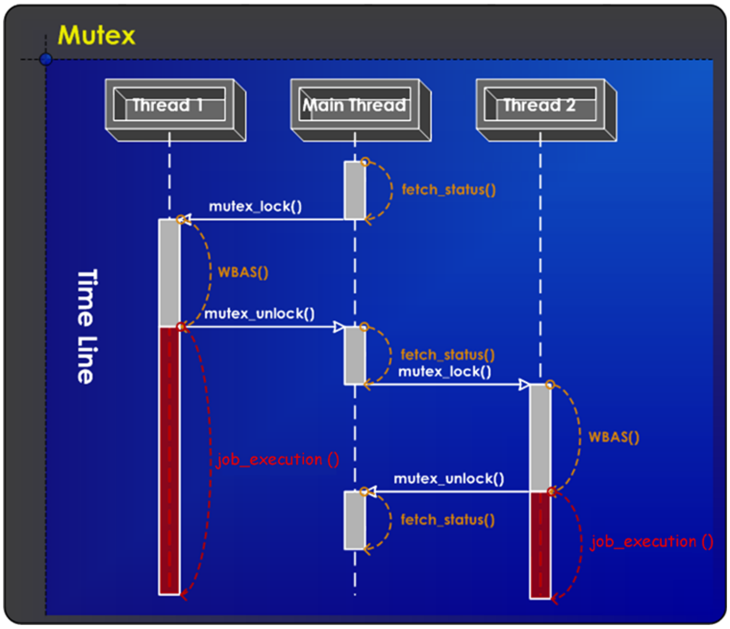
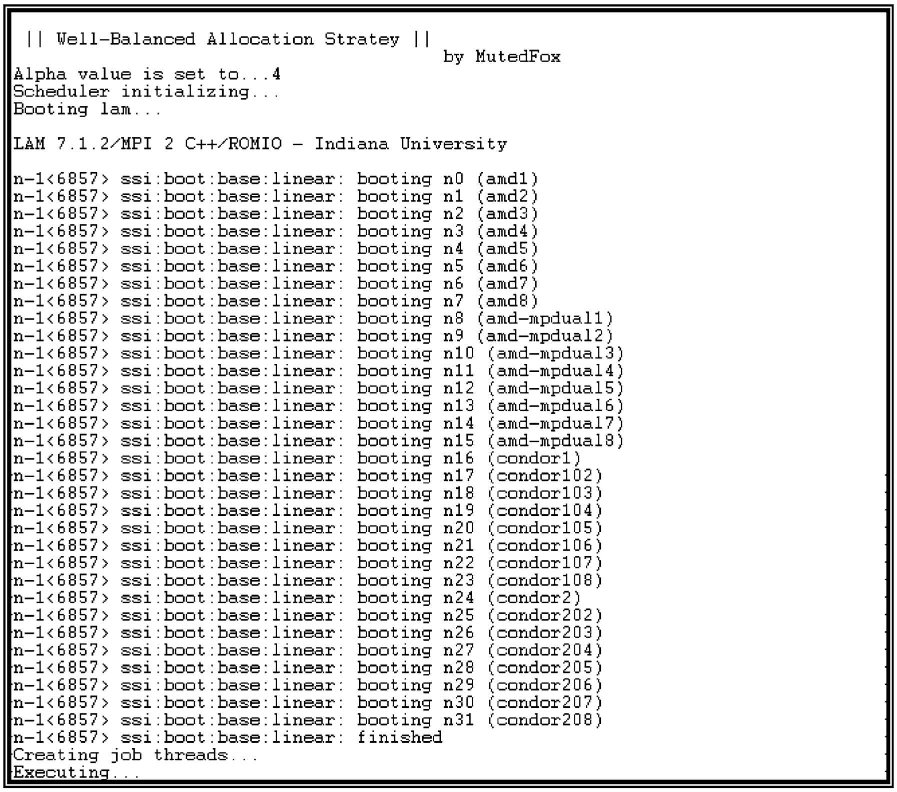


Fig. 13 Mutex example

The scheduling system is written in C++ language which is easier to implement under Linux. Figure 12 shows the main process flow of our scheduling system. After the scheduler has been executed, the main thread creates a number of job threads, where each thread stands for one job. The number of NP_{req}, which is the number of processors needed for this job, has also been generated. This job thread number could be of any size, based on how many jobs are allocated by this scheduler.

The next process is to allow these threads to compete for the critical area which is shown in Fig. 12. The scheduler uses mutual exclusion in this region, which only allows one thread to enter this part of the program. This mutex algorithm restricts simultaneous threads to use common resources in the critical areas. Figure 13 shows an example how the mutex work in our scheduler. In Fig. 13 we assume that we have a main thread and two job threads created on the scheduler. The time line is going from top to bottom.

At the beginning of the phase, the main thread first calls the fetch_status() function to fetch the status of all the nodes including the memory-free, one-minute load, five-minute load and fifteen-minute load. In our strategy, job thread one and job thread two compete with each other to enter the critical area where the mutex are. In this figure, it displays that thread one has won this round. Thread one is afterwards locked in the critical area by the mutex_lock() function. Now thread one may gain access to common resources, like files or memory spaces. As in our system, the WBAS() function is in the critical area for the scheduler to make scheduling decisions. The main purpose of the critical section is to protect the status of all machine states. In our scheduling system, we adjust the status of our machine before and after the execution of any job.



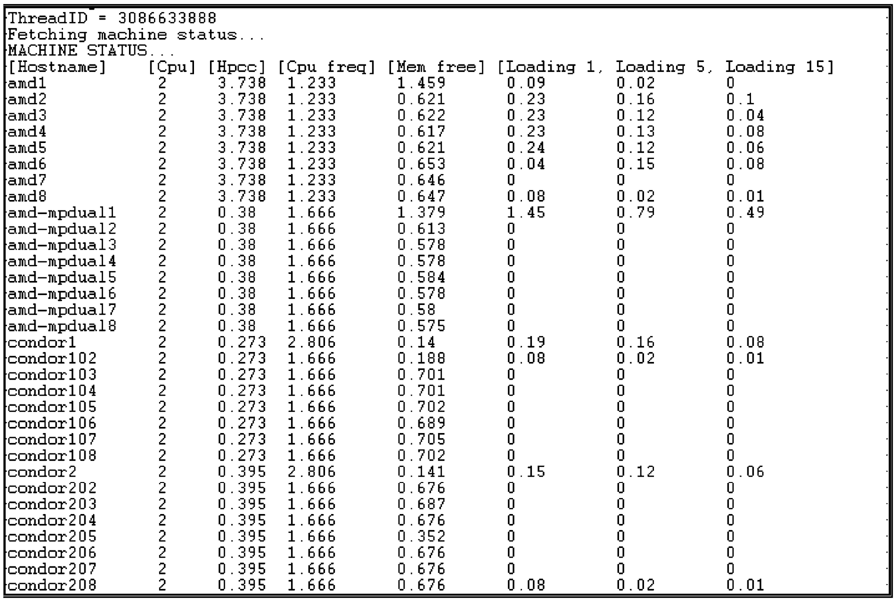
```
 || Well-Balanced Allocation Stratey ||
                                        by MutedFox
Alpha value is set to...4
Scheduler initializing...
Booting lam...

LAM 7.1.2/MPI 2 C++/ROMIO - Indiana University

n-1<6857> ssi:boot:base:linear: booting n0 (amd1)
n-1<6857> ssi:boot:base:linear: booting n1 (amd2)
n-1<6857> ssi:boot:base:linear: booting n2 (amd3)
n-1<6857> ssi:boot:base:linear: booting n3 (amd4)
n-1<6857> ssi:boot:base:linear: booting n4 (amd5)
n-1<6857> ssi:boot:base:linear: booting n5 (amd6)
n-1<6857> ssi:boot:base:linear: booting n6 (amd7)
n-1<6857> ssi:boot:base:linear: booting n7 (amd8)
n-1<6857> ssi:boot:base:linear: booting n8 (amd-mpdual1)
n-1<6857> ssi:boot:base:linear: booting n9 (amd-mpdual2)
n-1<6857> ssi:boot:base:linear: booting n10 (amd-mpdual3)
n-1<6857> ssi:boot:base:linear: booting n11 (amd-mpdual4)
n-1<6857> ssi:boot:base:linear: booting n12 (amd-mpdual5)
n-1<6857> ssi:boot:base:linear: booting n13 (amd-mpdual6)
n-1<6857> ssi:boot:base:linear: booting n14 (amd-mpdual7)
n-1<6857> ssi:boot:base:linear: booting n15 (amd-mpdual8)
n-1<6857> ssi:boot:base:linear: booting n16 (condor1)
n-1<6857> ssi:boot:base:linear: booting n17 (condor102)
n-1<6857> ssi:boot:base:linear: booting n18 (condor103)
n-1<6857> ssi:boot:base:linear: booting n19 (condor104)
n-1<6857> ssi:boot:base:linear: booting n20 (condor105)
n-1<6857> ssi:boot:base:linear: booting n21 (condor106)
n-1<6857> ssi:boot:base:linear: booting n22 (condor107)
n-1<6857> ssi:boot:base:linear: booting n23 (condor108)
n-1<6857> ssi:boot:base:linear: booting n24 (condor2)
n-1<6857> ssi:boot:base:linear: booting n25 (condor202)
n-1<6857> ssi:boot:base:linear: booting n26 (condor203)
n-1<6857> ssi:boot:base:linear: booting n27 (condor204)
n-1<6857> ssi:boot:base:linear: booting n28 (condor205)
n-1<6857> ssi:boot:base:linear: booting n29 (condor206)
n-1<6857> ssi:boot:base:linear: booting n30 (condor207)
n-1<6857> ssi:boot:base:linear: booting n31 (condor208)
n-1<6857> ssi:boot:base:linear: finished
Creating job threads...
Executing...
```

Fig. 14 Scheduler initializing state

Later on, if the decision has been made by the WBAS() function, then the job is dispatched to the selected nodes for execution, as shown in Fig. 15. On the other hand, if WBAS() failed to select the nodes, like if there are not sufficient nodes, thread one will have to make for another try to compete for the critical section. The critical area will be released by the mutex_unlock() function after the WBAS() function is finished. Then the main thread takes into control again to fetch the status of all the machines.

The figures below show the output of our scheduling simulator program. Three main phases are included in our process, scheduler initializing, fetch status, and job dispatch. Figure 14 shows the first phase of our scheduler. The α value is first shown on the screen, which is 4 in this figure. Then the scheduler is initialized by fetching the host information. Afterwards, LAM is booted by this host information, and the job threads are created at this point. The jobs are submitted and start to compete for the critical zone where the scheduling decision is made.

Figure 15 shows the status of all the machines in our multi-cluster pool. The job thread's ID which got into the critical zone is shown on the top. Then the scheduler commences to fetch the statuses on all the nodes. The columns Hostname, CPU, HPL value, and CPU frequency maintain the same value every time except when the



```
ThreadID = 3086633888
Fetching machine status...
MACHINE STATUS...
[Hostname]     [Cpu] [Hpcc] [Cpu freq] [Mem free] [Loading 1, Loading 5, Loading 15]
amd1           2     3.738  1.233      1.459      0.09       0.02       0
amd2           2     3.738  1.233      0.621      0.23       0.16       0.1
amd3           2     3.738  1.233      0.622      0.23       0.12       0.04
amd4           2     3.738  1.233      0.617      0.23       0.13       0.08
amd5           2     3.738  1.233      0.621      0.24       0.12       0.06
amd6           2     3.738  1.233      0.653      0.04       0.15       0.08
amd7           2     3.738  1.233      0.646      0          0          0
amd8           2     3.738  1.233      0.647      0.08       0.02       0.01
amd-mpdual1    2     0.38   1.666      1.379      1.45       0.79       0.49
amd-mpdual2    2     0.38   1.666      0.613      0          0          0
amd-mpdual3    2     0.38   1.666      0.578      0          0          0
amd-mpdual4    2     0.38   1.666      0.578      0          0          0
amd-mpdual5    2     0.38   1.666      0.584      0          0          0
amd-mpdual6    2     0.38   1.666      0.578      0          0          0
amd-mpdual7    2     0.38   1.666      0.58       0          0          0
amd-mpdual8    2     0.38   1.666      0.575      0          0          0
condor1        2     0.273  2.806      0.14       0.19       0.16       0.08
condor102      2     0.273  1.666      0.188      0.08       0.02       0.01
condor103      2     0.273  1.666      0.701      0          0          0
condor104      2     0.273  1.666      0.701      0          0          0
condor105      2     0.273  1.666      0.702      0          0          0
condor106      2     0.273  1.666      0.689      0          0          0
condor107      2     0.273  1.666      0.705      0          0          0
condor108      2     0.273  1.666      0.702      0          0          0
condor2        2     0.395  2.806      0.141      0.15       0.12       0.06
condor202      2     0.395  1.666      0.676      0          0          0
condor203      2     0.395  1.666      0.687      0          0          0
condor204      2     0.395  1.666      0.676      0          0          0
condor205      2     0.395  1.666      0.352      0          0          0
condor206      2     0.395  1.666      0.676      0          0          0
condor207      2     0.395  1.666      0.676      0          0          0
condor208      2     0.395  1.666      0.676      0.08       0.02       0.01
```

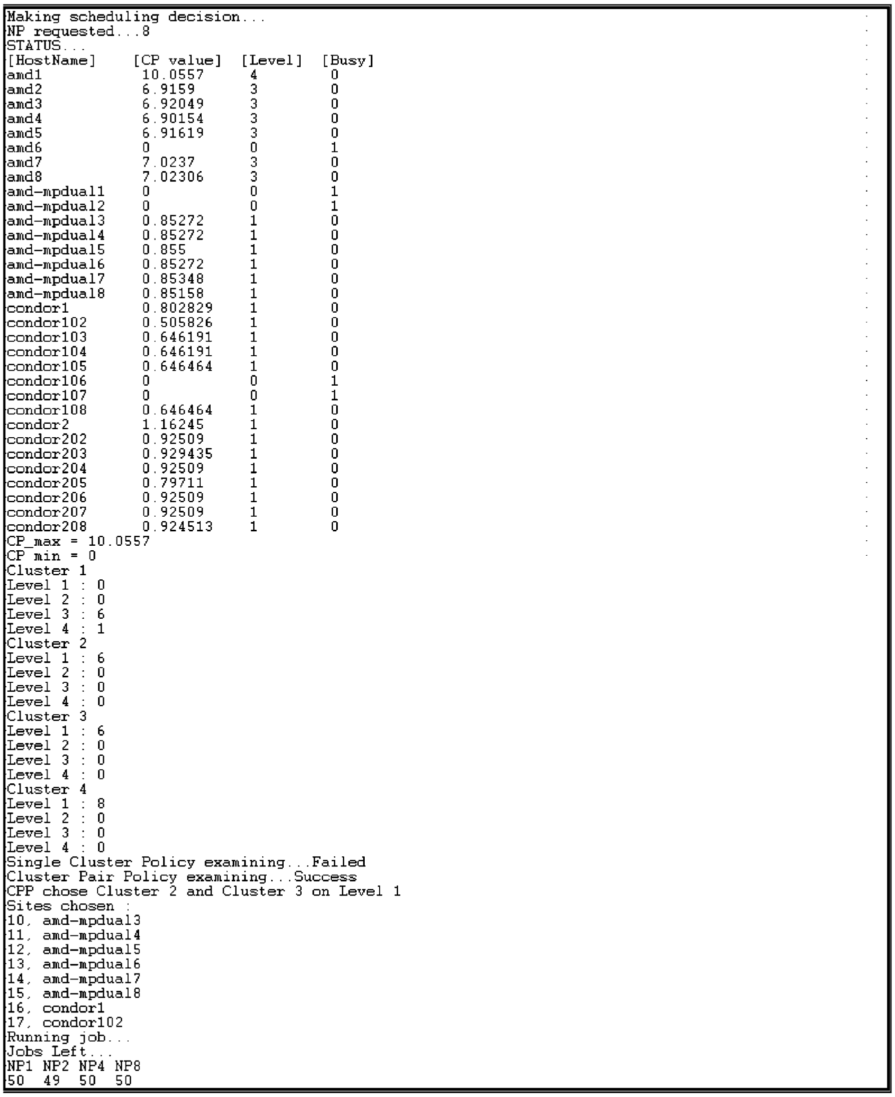
Fig. 15 Fetch status state

hardware is changed on certain nodes or clusters. The values of memory-free, one-minute load, five-minute load, and fifteen-minute load are fetched at this moment.

Figure 16 shows how the job is dispatched by our WBAS policy. The second line displays the number of requested nodes for this job. Then the *CP* value of each node is calculated by the information given in Fig. 16. These nodes are classified into α levels which were set to 4 in this example. Afterwards, the $CP_{\max}$, $CP_{\min}$, and the number of nodes of each level under certain cluster are also outputted onto the screen. In addition, the nodes which are busy are set to be busy state, the CP values of these are not calculated and directly set to zero. We can see that the single-cluster policy failed to select the nodes because we do not have sufficient nodes. Nevertheless, the cluster-pair policy succeeded choosing size nodes from cluster 2 and two nodes from cluster 3. Eventually, the job is dispatched onto these selected nodes for execution. The leftover jobs are shown in the bottom of Fig. 16.

4 Experimental results

Our multi-cluster environment includes four clusters with eight nodes, including one head node and seven tail nodes. These machines all participate in job execution. The master node is responsible for dispatching jobs to the clusters and for evaluating performance. The hardware and software specifications of our multi-cluster environment are shown in Table 2. The network bandwidths and the environment are shown in Fig. 17. The bandwidths are all 10/100M, except for cluster "amd1," where the nodes are the fastest. This means that its HPL value is much higher than those of the



```
Making scheduling decision...
NP requested...8
STATUS...
[HostName]    [CP value]  [Level]  [Busy]
amd1           10.0557     4        0
amd2           6.9159      3        0
amd3           6.92049     3        0
amd4           6.90154     3        0
amd5           6.91619     3        0
amd6           0           0        1
amd7           7.0237      3        0
amd8           7.02306     3        0
amd-mpdual1    0           0        1
amd-mpdual2    0           0        1
amd-mpdual3    0.85272     1        0
amd-mpdual4    0.85272     1        0
amd-mpdual5    0.855       1        0
amd-mpdual6    0.85272     1        0
amd-mpdual7    0.85348     1        0
amd-mpdual8    0.85158     1        0
condor1        0.802829    1        0
condor102      0.505826    1        0
condor103      0.646191    1        0
condor104      0.646191    1        0
condor105      0.646464    1        0
condor106      0           0        1
condor107      0           0        1
condor108      0.646464    1        0
condor2        1.16245     1        0
condor202      0.92509     1        0
condor203      0.929435    1        0
condor204      0.92509     1        0
condor205      0.79711     1        0
condor206      0.92509     1        0
condor207      0.92509     1        0
condor208      0.924513    1        0
CP_max = 10.0557
CP min = 0
Cluster 1
Level 1 : 0
Level 2 : 0
Level 3 : 6
Level 4 : 1
Cluster 2
Level 1 : 6
Level 2 : 0
Level 3 : 0
Level 4 : 0
Cluster 3
Level 1 : 6
Level 2 : 0
Level 3 : 0
Level 4 : 0
Cluster 4
Level 1 : 8
Level 2 : 0
Level 3 : 0
Level 4 : 0
Single Cluster Policy examining...Failed
Cluster Pair Policy examining...Success
CPP chose Cluster 2 and Cluster 3 on Level 1
Sites chosen :
10, amd-mpdual3
11, amd-mpdual4
12, amd-mpdual5
13, amd-mpdual6
14, amd-mpdual7
15, amd-mpdual8
16, condor1
17, condor102
Running job...
Jobs Left...
NP1 NP2 NP4 NP8
50  49  50  50
```

Fig. 16 Job dispatch state

other three. “amd-mpdual1,” “condor1,” and “condor2” perform nearly as well, but they run in diskless environments.

The clusters are linked to a switch that connects to the master node where the scheduler and monitoring system are located. The master node is responsible for submitting jobs to the clusters, and fetching and storing the statuses of all machines. The head and tail nodes are responsible for executing jobs in this experimental environment, and return their statuses and outputs to the master node. The master node is responsible for fetching the statuses of all the machines and submitting the jobs onto the clusters, and also keeping the state of all the nodes. The head and tail nodes are

Table 2 Hardware and software specifications

	amd1		amd-mpdual1		condor1		condor2	
	H	T	H	T	H	T	H	T
PC#	1	7	1	7	1	7	1	7
CPU	AMD Athlon MP 2600+		AMD Athlon MP 2000+		Intel P4 2.8 GHz	AMD Athlon MP 2000+	Intel P4 2.8 GHz	AMD Athlon MP 2000+
RAM	2 GB	1 GB	2 GB	1 GB	2 GB	512 KB	2 GB	1 GB
Hard Disk	Yes				Yes	No	Yes	No
NIC	Gigabit				10/100M			
OS	Fedora Core 5				Fedora Core 6			
Lam	lam-7.1.2-1.fc5				lam-7.1.2-8.fc6			

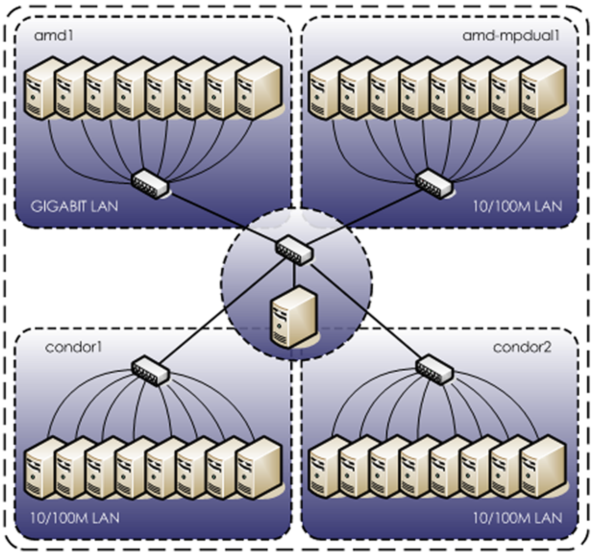


Fig. 17 Network bandwidths

basically compute nodes in this experimental environment for returning the information before execution and output back to the master node when the job is done.

The first step is to obtain the HPL value of each cluster. Therefore, we gather this value by executing the benchmark program once only on them, as shown in Table 3. After calculating and storing the HPL values for all clusters, we submitted ten runs of two hundred MPI-based jobs, fifty requesting just one node, fifty requesting two nodes, fifty requesting four nodes, and fifty requesting eight nodes, and measured the average response times. We compared experimental results with sequential execution in which the jobs are executed one by one, to demonstrate the speedup improvement. In the following figures, “WBAS 3” means that the number of levels is 3 (i.e., α is 3).

The benchmarks are matrix multiplication MPI programs in which the matrix size could be varied. The results in Fig. 18 show that when the size was set to 512, the WBAS performed best with the α value set to 4, and took a long time to finish when it

Table 3 HPL values of each cluster

	amd1	amd-mpdual1	condor1	condor2
HPL (GFlops)	29.9	3.86	2.181	3.317

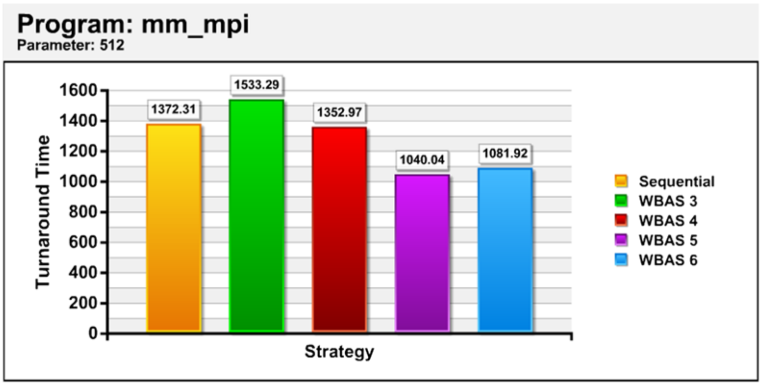


Fig. 18 Results for mm_mpi 512

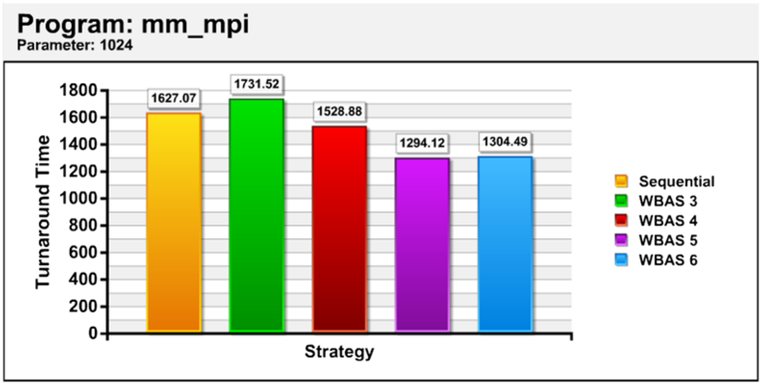


Fig. 19 Results for mm_mpi 1024

was set to 3. "Sequential" took even longer. We then set the matrix size to 1024, and the results in Fig. 19 show that the system performed almost exactly as it did when the matrix size was set to 512, with only slight increase in time. Figure 20 shows the results obtained when the matrix size was set to be 2048. The nodes took a much greater amount of time to finish the jobs. Note that in this situation, the shortest time was recorded when the α value was set to 3. The "sequential" strategy, on the other hand, finished almost half an hour later.

The next program calculates the prime numbers below a certain value, which could be changed by the user. The total prime number and the largest are computed. Figure 21 shows that the parameter is set to 20 million; we can see that "sequential"

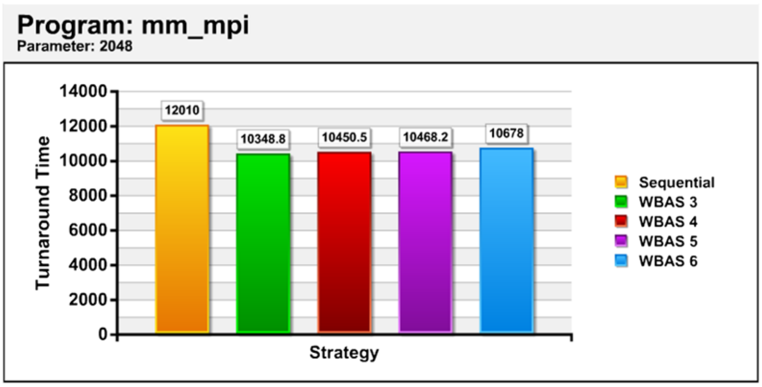


Fig. 20 Results for mm_mpi 2048

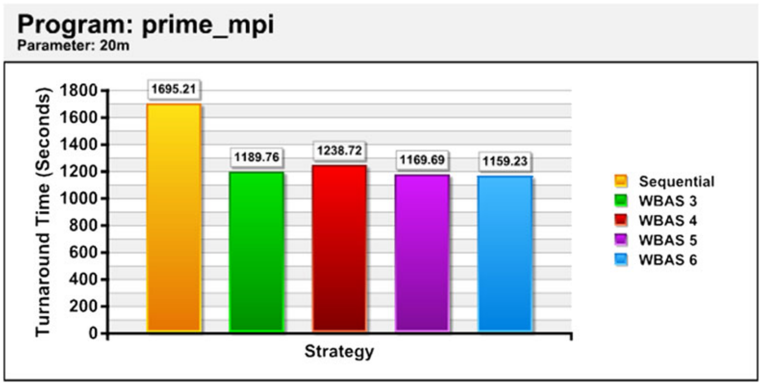


Fig. 21 Results for prime_mpi 20m

behaves the worst in this case. In Fig. 22 that parameter is set to 50 million. The results display that "WBAS 5" has the best result, but the turnaround time is very similar in each case. The results shown in Fig. 23 are rather close when setting the parameter to 100 million. The finishing time between "sequential" and "WBAS 5" is about 10 minutes.

The next experiment calculates the π which is the ratio of any circle's circumference to its diameter in Euclidean geometry, as shown in Fig. 24. Our strategy had the biggest impact on jobs that require a long time to finish. Because faster nodes of cluster have to wait for slower nodes to finish, the idle time is wasted since the resource is unused.

Figure 25 shows the program to solve the classical N-body problem which simulates the evolution of a system of N bodies and where the force exerted on each body arises due to its interaction with all the other bodies in the system. The performance is rather close from "sequential" to "WBAS" with different α values. Figure 26 shows a sequential satisfiability program in which the task is to give a logical formula in sixteen variables to determine which values of the inputs make the formula true. The

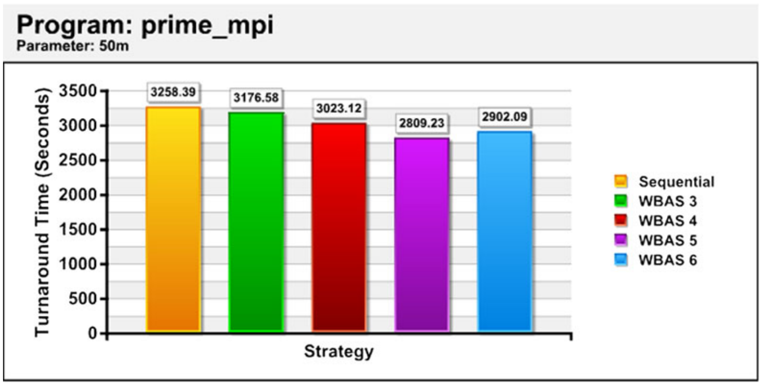


Fig. 22 Results for prime_mpi 50m

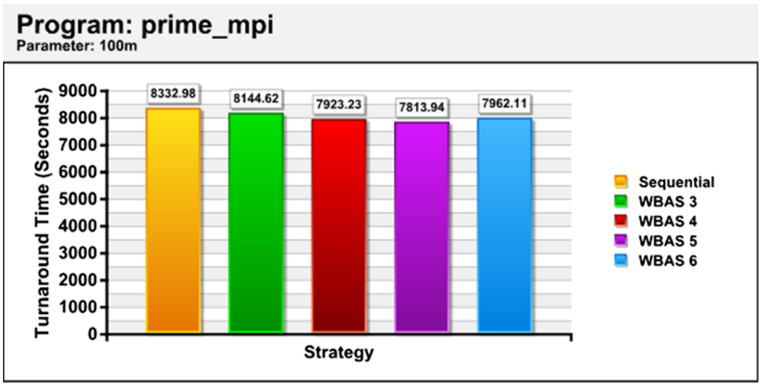


Fig. 23 Results for prime_mpi 100m

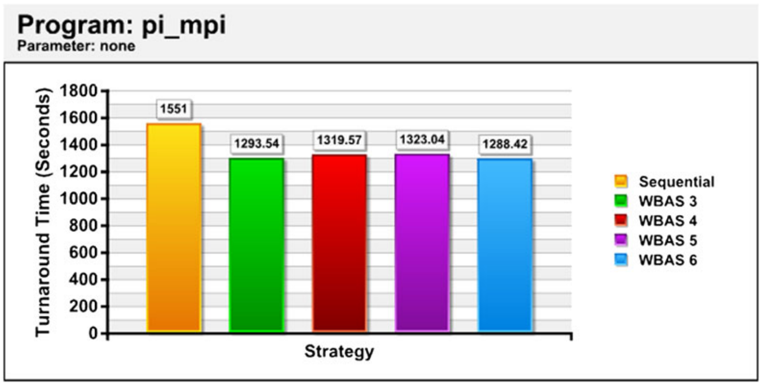


Fig. 24 Results for pi_mpi

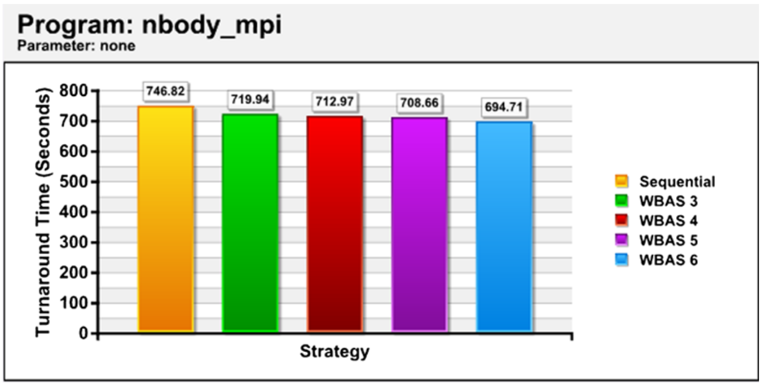


Fig. 25 Results for nbody_mpi

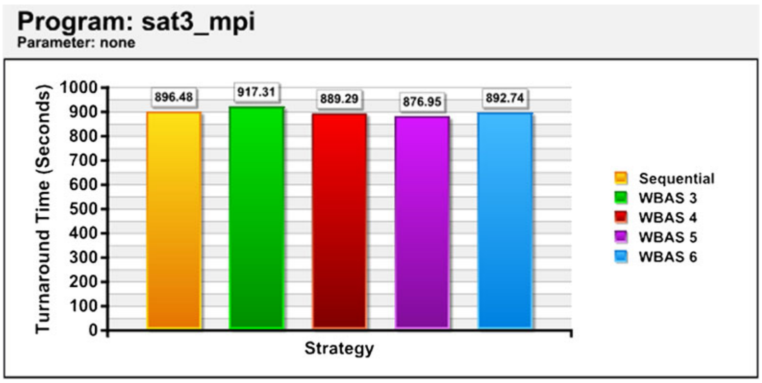


Fig. 26 Results for sat3_mpi

figure also shows that the performances are still close to each other from the last experiment for which both of these programs only take less time to finish.

Figure 27 shows a program for solving the N-queen problem. The original problem is called the eight-queen problem which is trying to place eight queens on a chess board so that no queen could attack any other queen. The N-queen problem expands this concept into N by N with variable input in this MPI program. This figure shows the result of setting the parameter to 10 due to the chess board having the size of 10 by 10. It behaves rather well under this kind of problem size, the method of WBAS with α set to 5 finished about one minute earlier than the sequential method.

Our strategy had the biggest impact on jobs that require a long time to finish. Because of faster nodes of cluster having to wait for slower nodes to finish, the idle time is wasted as the resource is unused.

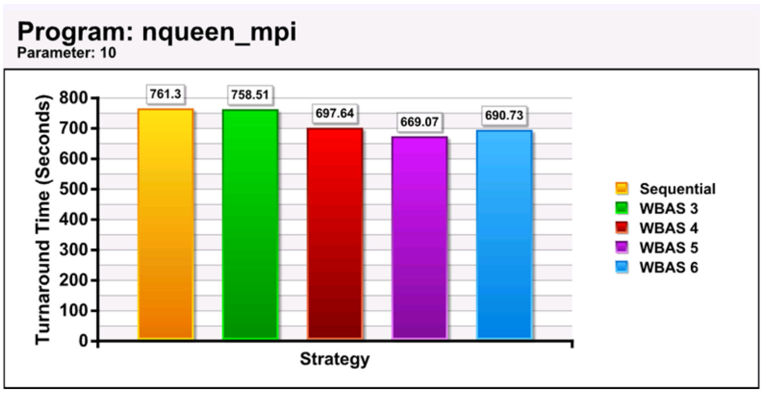


Fig. 27 Results for nqueen_mpi 10

5 Conclusion and future work

In building cluster computing platforms, we commonly interconnect multiple personal computers or workstations to achieve high computing power. However, picking the right tool to make the best use of our clusters is critical. The proposed work enables users to exploit all available resources in multi-cluster environments. As well as monitoring all nodes in the system, Ganglia also provides an advanced web front-end. We can check node availability, and the status of all machines. With Condor, users can easily submit jobs to and remove them from the queue.

The proposed WBAS could distribute loads across all the machines. It works on homogeneous as well as heterogeneous multi-clusters, and even on non-dedicated machines. It takes CPU, free memory, loadings, and network latency into account. Moreover, HPL value, the benchmark used to evaluate the power of clusters around the world, is also considered. Users need only to specify the number of required nodes, and the scheduler does the matchmaking automatically, deploying jobs to the most suitable nodes. The nodes will finish executing the task at nearly the same time to avoid excessive idling of the system.

Our future work may be in enhancing the proposed scheduling strategy. Static scheduling could be combined with dynamic scheduling in order to adjust node loadings during execution, or even to predict the job execution time in advance for improving the system performance.

Acknowledgement This work is supported in part by National Science Council, Taiwan R.O.C., under grants No. NSC 96-2221-E-029-019-MY3, NSC 98-2218-E-007-005 and NSC 98-2622-E-029-001-CC2.

References

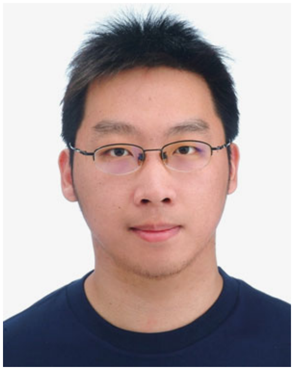
1. Abawajy JH (2009) An efficient adaptive scheduling policy for high-performance computing. Future Gener Comput Syst 25(3):364–370
2. Anderson T, Culler D, Patterson D (1995) A case for network of workstations. IEEE Micro 15(1):54–64

3. Buyya R (1999) High performance cluster computing: system and architectures, vol 1. Prentice Hall, New York
4. Buyya R (1999) High performance cluster computing: programming and applications, vol 2. Prentice Hall, New York
5. Cao J, Chan A, Sun Y, Das SK, Guo M (2006) A taxonomy of application scheduling tools for high performance cluster computing. J Clust Comput 9(3):355–371
6. Chen DZ, Wang YM (2007) The impact of memory resource on loop self-scheduling for heterogeneous clusters. In: CTHPC 2007
7. Bucur AID, Epema DHJ (2007) Scheduling policies for processor coallocation in multicluster systems. IEEE Trans Parallel Distrib Syst 18(7):958–972
8. Foster I, Kesselman C (1999) The grid: blueprint for a future computing infrastructure. Morgan Kaufmann, San Mateo
9. Foster I, Kesselman C, Tuecke S (2001) The anatomy of the grid: Enabling scalable virtual organizations. Int J Supercomput Appl 15(3)
10. Geist A (1994) Cluster computing: the wave of the future. Lecture notes in computer science, vol 879. Springer, Berlin, pp 236–246
11. Jones WM, Ligon III WB, Pang L.W., Stanzione D. (2005) Characterization of bandwidth-aware meta-schedulers for co-allocating jobs across multiple clusters. J Supercomput 34(2):135–163
12. Krueger PE, Livny M (1988) A comparison of preemptive and non-preemptive load distributing. In: Proc of the 8th international conference on distributed computing systems, pp 123–130, June 1988
13. Matsuda M, Kudoh T, Ishikawa Y (2003) Evaluation of MPI implementations on grid-connected clusters using an emulated WAN environment. In: Proc of the 3rd IEEE/ACM international symposium on cluster computing and the grid (CCGRID'03). IEEE Computing Society, p 10
14. Mutka M, Livny M (1987) Scheduling remote processing capacity in a workstation-processing bank computing system. In: Proceedings of the 7th international conference of distributed computing systems, pp 2–9, September, 1987
15. Silberstein M, Geiger D, Schuster A, Livny M (2006) Scheduling mixed workloads in multi-grids: the grid execution hierarchy. In: Proceedings of the 15th IEEE symposium on high performance distributed computing (HPDC), pp 33–40
16. Sterling TL, Salmon J, Backer DJ, Savarese DF (1999) How to build a beowulf: a guide to the implementation and application of PC clusters, 2nd edn. MIT, Cambridge
17. Wang Y-M (2006) Memory latency consideration for load sharing on heterogeneous network of workstations. J Syst Archit, EUROMICRO J 52(1):13–20
18. Werstein P, Situ H, Huang Z (2006) Load balancing in a cluster computer. In: Proceedings of the seventh international conference on parallel and distributed computing, applications and technologies, pp 569–577
19. Wilkinson B, Allen M (1999) Parallel programming: techniques and applications using networked workstations and parallel computers. Prentice Hall, New York, 1999
20. Wright D (2001) Cheap cycles from the desktop to the dedicated cluster: Combining opportunistic and dedicated scheduling with Condor. In: Conference on Linux clusters: the HPC revolution, June 2001
21. Xavier P, Cai W, Lee BS (2006) Workload management of cooperatively federated computing clusters. J Supercomput 36(3):309–322
22. Yang CT, Chang SC (2004) A parallel loop self-scheduling on extremely heterogeneous PC clusters. J Inf Sci Eng 20(2):263–273
23. Yang CT, Chen PI, Chen YL (2005) Performance evaluations of SLIM and DRBL diskless PC clusters on Fedora Core 3. In: Proceedings of the 6th IEEE international conference on parallel and distributed computing, applications and technologies (PDCAT 2005), pp 479–482, December 5–8, 2005
24. Yang CT, Liao CS, Chen PI, Tung HY (2006) An information monitoring and job scheduling system for multiple Linux PC clusters. In: Proceedings of the 7th international conference on parallel and distributed computing, applications and technologies (PDCAT 2006), IEEE CS Press, pp 578–582, Taipei, Taiwan, December 4–7, 2006
25. Yang CT, Chen PI, Chen SY, Tung HY (2006) A jobs' allocation strategy for multiple DRBL diskless Linux clusters with Condor schedulers. In: Proceedings of the 5th international conference on grid and cooperative computing (GCC 2006), IEEE CS Press, pp 54–57, China, Oct 2006
26. Yang CT, Chen PI, Hu YC, Tung HY, Ke C-C (2006) On utilization of multiple DRBL-based Linux clusters in the computer classroom to grid computing environments. In: Proceedings of the 12th workshop on compiler techniques for high-performance computing (CTHPC 2006), pp 36–41, Tainan, Taiwan, March 16–17, 2006

27. Yang CT, Chen TT, Tung HY (2007) A dynamic domain-based network information model for computational grids. In: Future generation communication and networking (FGCN 2007), pp 575–578, Jeju-Island, Korea, December 6–8, 2007
28. MPI Forum (1994) MPI: A message-passing interface standard. Int J Supercomput Appl 8(3/4):165–416
29. DRBL, http://drbl.sourceforge.net/
30. Ganglia, http://ganglia.info/
31. LAM/MPI Parallel Computing, http://www.lam-mpi.org/
32. Message Passing Interface Forum, http://www.mpi-forum.org/
33. MPICH, http://www-unix.mcs.anl.gov/mpi/mpich1/
34. PVM—Parallel Virtual Machine, http://www.epm.ornl.gov/pvm
35. Arabnia HR, Oliver MA (1987) Arbitrary rotation of raster images with SIMD machine architectures. Int J Eurographics Assoc, Comput Graph Forum 6(1):3–12
36. Bhandarkar SM, Arabnia HR, Smith JW (1995) A reconfigurable architecture for image processing and computer vision. Int J Pattern Recognit Artif Intell 9(2):201–229 (special issue on VLSI Algorithms and Architectures for Computer Vision, Image Processing, Pattern Recognition and AI)
37. Bhandarkar SM, Arabnia HR (1995) The Hough transform on a reconfigurable multi-ring network. J Parallel Distrib Comput 24(1):107–114

Chao-Tung Yang is a Professor of Computer Science at Tunghai University in Taiwan. He was born on November 9, 1968 in Ilan, Taiwan, R.O.C. and received a B.Sc. degree in Computer Science from Tunghai University, Taichung, Taiwan, in 1990, and the M.Sc. degree in Computer Science from National Chiao Tung University, Hsinchu, Taiwan, in 1992. He received the Ph.D. degree in Computer Science from National Chiao Tung University in July 1996. He won the 1996 Acer Dragon Award for an outstanding Ph.D. dissertation. He has worked as an Associate Researcher for ground operations in the ROCSAT Ground System Section (RGS) of the National Space Program Office (NSPO) in Hsinchu Science-based Industrial Park since 1996. In August 2001, he joined the Faculty of the Department of Computer Science at Tunghai University. He got the Excellent Research Award from Tunghai University in 2007. In 2007 and 2008, he got the Golden Penguin Award by Industrial Development Bureau, Ministry of Economic Affairs, Taiwan. His researches have been sponsored by Taiwan agencies of National Science Council (NSC), National Center for High Performance Computing (NCHC), and Ministry of Education. His present research interests are in grid and cluster computing, parallel and multi-core computing, and Web-based applications. He is both a member of the IEEE Computer Society and ACM.

Kuan-Chou Lai received his M.Sc. degree in Computer Science and Information Engineering from the National Cheng Kung University in 1991, and the Ph.D. degree in Computer Science and Information Engineering from the National Chiao Tung University in 1996. Currently, he is an Associate Professor in the Department of Computer and Information Science and the director of the Computer and Network Center at the National Taichung University. His research interests include parallel processing, heterogeneous computing, system architecture, P2P, grid computing, and multimedia systems. He is a member of the IEEE and the IEEE Computer Society.



Hao-Yu Tung was born on July 5th, 1983, in Taichung City, Taiwan, R.O.C. He received the B.S. degree in Department of Computer Science from Tunghai University in 2006, and he also received the M.S. degree in Department of Computer Science from Tunghai University as well in 2008. He is currently an engineer of HTC Corporation in Xindian City now.