Dynamic Scheduling of Parallel Jobs with QoS Demands in Multiclusters and Grids

Ligang He, Stephen A. Jarvis, Daniel P. Spooner, Xinuo Chen and Graham R. Nudd
Department of Computer Science, University of Warwick,
Coventry, United Kingdom, CV4 7AL
liganghe@dcs.warwick.ac.uk

Abstract

This paper addresses the dynamic scheduling of parallel jobs with QoS demands (soft-deadlines) in multiclusters and grids. Three metrics (over-deadline, makespan and idle-time) are combined with variable weights to evaluate the scheduling performance. These three metrics are used to measure the extent of the jobs' QoS demand compliance, the resource throughput and the resource utilization. Two levels of performance optimisation are applied in the multicluster. At the multicluster level, a scheduler (which we call MUSCLE) allocates parallel jobs with high packing potential to the same cluster; it also takes the jobs' QoS requirements into account and employs a heuristic to allocate suitable workloads to each cluster to balance the overall system performance. At the single cluster level, an existing workload manager, called TITAN, utilizes a genetic algorithm to further improve the scheduling performance of the jobs previously allocated by MUSCLE. Extensive experimental studies are conducted to verify the effectiveness of the scheduling mechanism as well as the effect of the prediction accuracy on the scheduling performance. The results show that compared with traditional distributed workload allocation policies, the comprehensive scheduling performance of parallel jobs is significantly improved across the multicluster, and the presence of prediction errors does not dramatically weaken the performance advantage.[*]

1. Introduction

Separate clusters are increasingly being interconnected to create multiclusters or grid computing architectures. The constituent clusters may be located within a single organization or across different administrative organizations [1][2]. Workload management and scheduling are key issues in grid computing, and parallel jobs constitute a typical workload in the scheduling scenario. Parallel jobs can be classified into two categories: rigid and moldable [12]. Rigid parallel jobs are run on a user-specified number of computational resources, while moldable jobs can be run on different numbers of resources. This paper investigates rigid jobs.

Qualities of service (QoS) are often associated with jobs submitted to a grid system [2][9]. An example of this is jobs with user-defined deadlines [2]. The QoS of a job is satisfied if it finishes before the specified deadline, while the QoS decreases as the excess (of completion time over deadline) increases. Therefore, over-deadline can be used to measure the extent to which the QoS demands of a set of jobs are satisfied. Over-deadline is defined as the sum of excess time of each job's finish time over its deadline. The scheduling of parallel jobs has been extensively studied in single cluster environments [10] [12]. In such a scenario, *resource utilization* is a commonly used system-oriented metric [12], which is often measured by *idle-time* in resources [12]. An additional major goal in job scheduling in grids is high *resource throughput*. *Makespan* is a commonly used metric to measure resource throughput [9]. Makespan is defined as the duration between the start time of the first job and the finish time of the last executed job.

In the multicluster architecture assumed in this paper, the constituent clusters may be located in different administrative organizations and as a result be managed with different performance criteria. In this scheduling work, three metrics (over-deadline, makespan and idle-time) are combined with variable weights to evaluate scheduling performance, which allows the resources in different locations to represent different performance scenarios.

Tightly packing parallel jobs into resources is beneficial to achieving high resource utilization and throughput [12][13]. To some extent, it also benefits the reduction of over-deadline. However, overemphasizing the packing may comprise performance improvements in terms of over-deadline. It is therefore necessary to find the schedule that offers high comprehensive performance accord-

[*] This work is sponsored in part by grants from the NASA AMES Research Center (administrated by USARDSG, contract no. N68171-01-C-9012), the EPSRC (contract no. GR/R47424/01) and the EPSRC e-Science Core Programme (contract no. GR/S03058/01).

ing to the performance requirement of each individually managed resource.

In this work, the multicluster is equipped with two levels of performance optimisation. A multicluster-level scheduler (called MUSCLE) allocates parallel jobs with high packing potential (i.e., they can be packed more tightly) to the same cluster; it also takes the jobs' QoS demands into account and employs a heuristic to control workload allocation among clusters, so as to balance the overall performance across the multicluster. When MUSCLE distributes jobs to clusters, it also determines a *seed schedule* for the jobs allocated to each cluster. At the local cluster level, an existing workload manager (TITAN [13]) receives the seed schedule and uses a genetic algorithm to transform the schedule into one that improves the (local) comprehensive performance (in terms of over-deadline, makespan and idle-time).

Existing scheduling mechanisms can be classified into two categories: on-line mode and batch mode [9]. Using an on-line mode technique, a job is scheduled to resources as soon as it arrives. In batch mode, however, arriving jobs are collected for examination at prescheduled times and these jobs are scheduled as a meta-task when a scheduling event is triggered [9]. The batch mode technique can have more job information than the on-line mode technique when making scheduling decisions [9][12]. The scheduling mechanism in MUSCLE belongs to the batch mode category. MUSCLE schedules a batch of collected jobs to individual clusters. Then MUSCLE collects and organizes the subsequently arriving jobs; while at each individual cluster, the jobs previously allocated by MUSCLE are scheduled (by TITAN) for final execution. When the expected makespan of the jobs yet to be scheduled is less than a predefined threshold (determined by TITAN), a new round of scheduling is performed by MUSCLE for the newly collected jobs.

Job scheduling is extensively documented [2][4][6][8][9]. Mechanisms of co-location and adaptive scheduling are presented in AppleS [4]. However, AppleS does not consider a job' deadline when scheduling. Nimrod [2] takes the jobs' deadlines into account. However, it is based on an economy model and therefore has different concerns from this work. A QoS guided Min-Min heuristic is presented in [9] for grids. The QoS in [9] is based on the network bandwidth, and makespan is used to evaluate the scheduling performance. In this paper, multiple performance metrics are considered.

The rest of the paper is organized as follows. The system and workload model is introduced in Section 2. Section 3 presents the genetic algorithm used in TITAN. The scheduling mechanism of MUSLCE is proposed in Section 4. Section 5 presents experimental results and Section 6 concludes the paper.

2. System and workload model

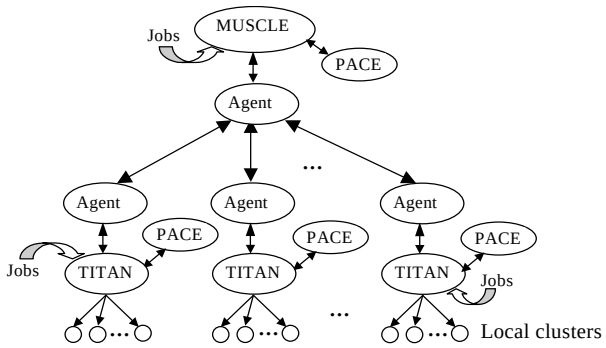


Figure 1. **The multicluster architecture**

The multicluster in this work (shown in Figure 1) consists of n clusters, C_1, C_2, …, C_n; where a cluster C_i ($1 \le i \le n$) consists of m_i homogeneous computational resources (the size of cluster C_i is m_i), each with a service rate of u_i. There are two scheduling levels in the architecture; MUSCLE acts as the global scheduler for the multicluster while TITAN further transforms and improves the schedules sent by MUSCLE within each local cluster. MUSCLE and TITAN are interconnected through an agent system called A4 (Agile Architecture and Autonomous Agents) [3]. Users can submit parallel jobs to the multicluster through MUSCLE or through TITAN. If a job is submitted via TITAN, MUSCLE is made aware of it through metadata sent by the A4 agents. When the makespan of the jobs yet to be scheduled in the clusters is less than a predefined threshold, MUSCLE is also made aware of this via A4 and enters a new round of job scheduling. The PACE (Performance Analysis and Characterisation Environment) toolkit [11] is used to predict the jobs' execution time prior to run time. This paper focuses on presenting the scheduling mechanism in MUSCLE.

Parallel jobs considered in this paper are rigid. A parallel job, denoted by J_i, is identified by a 4-tuple (a_i, s_i, e_{ij}, d_i), where a_i is J_i's arrival time, s_i is its job size (i.e. the number of computational resources on which the job is requested to be run), e_{ij} is its execution time in cluster C_j ($1 \le j \le n$) and d_i is its soft-deadline.

3. Genetic algorithm in TITAN

This section briefly describes the genetic algorithm used in the local TITAN workload manager [13]. A two dimensional coding scheme is developed to represent a schedule of parallel jobs in a cluster. Each column in the coding specifies the allocation of the processing nodes to a parallel job, while the order of these columns in the coding is also the order in which the corresponding jobs are to be executed.

Three metrics (over-deadline, makespan and idle-time) are combined with weights to form a comprehensive per-

formance metric (denoted by *CP*), which is used to evaluate a schedule. CP is defined in Eq.1, where Γ, ω and θ are idle-time, makespan and over-deadline, respectively; and W^i, W^m and W^o are their weights. For a given weight combination, the lower the value of CP, the better the comprehensive performance.

$$CP = \frac{W^i\Gamma + W^m\omega + W^o\theta}{W^i + W^m + W^o} \quad (1)$$

A genetic algorithm is used in TITAN to find a schedule with a low CP. The algorithm first generates a set of initial schedules (one of these is the seed schedule sent by MUSCLE). *Crossover* and *mutation* are performed to transform the schedules in the current set and generate the next generation. This procedure continues until the performance in each generation of schedule stabilizes.

4. Scheduling mechanism in MUSCLE

The main operations performed by MUSCLE are as follows. First, MUSCLE determines which parallel jobs can be packed into a *resource space* of a given size (i.e., available resources of a given number). It is possible that in a set of parallel jobs, different compositions of jobs can be packed into a resource space. These possible compositions of parallel jobs are organized into a *composition table*. The *i*-th row of the table corresponds to the possible compositions of parallel jobs for packing into a resource space of size *i*.

The constructed composition table is used to support job allocation. MUSCLE utilizes a heuristic to determine the cluster to which jobs should be allocated. Then MUSCLE finds the earliest available resource space in that cluster, and searches the composition table for suitable jobs to allocate to the space. After that, the job allocation status in that cluster is updated and the heuristic is invoked again for allocating further jobs.

4.1. Organizing parallel jobs

Suppose the maximum cluster size in the multicluster is m_{MAX} and that p parallel jobs, J_1, J_2, …, J_p are collected by MUSCLE (into a queue). Algorithm 1 outlines the steps for constructing the composition table. The p jobs are filled into suitable rows in the table. When trying to fill job J_i (with size s_i) into the j-th row, the algorithm checks if there exists such a composition in the $(j\text{-}s_i)^{\text{th}}$ row that no job in the composition has appeared in the j-th row. If such a composition exists, it indicates that J_i and the jobs in the composition can be packed into the resource space with size j. Hence, J_i and these jobs are filled into this row.

Algorithm 1. Constructing the composition table

1. **for** each parallel job J_i ($1 \le i \le p$) to be scheduled **do**
2. **for** each j satisfying $1 \le j \le m_{MAX}$ **do**
3. **if** $s_i = j$
4. Append J_i to the tail of the j-th row in the table;
5. **if** $s_i < j$
6. $r \leftarrow j - s_i$;
7. **if** the r-th row in the table is not NULL
8. **if** there is such a composition of parallel jobs in the r-th row, in which no job appears in the j-th row of the table;
9. Append J_i as well as the parallel jobs in the composition from the r-th row to the tail of the j-th row;

Table 1. **An example composition table; the jobs in the queue are J_1-J_6, where the sizes of J_1-J_6 are 2, 1, 4, 3, 1 and 2, respectively; m_{MAX} is 6**

1	J_2	J_5	
2	J_1	J_5, J_2	J_6
3	J_2, J_1	J_4	J_6, J_5
4	J_3	J_4, J_2	J_6, J_1
5	J_3, J_2	J_4, J_1	
6	J_3, J_1	J_6, J_4, J_2	

There are two for-loops in Algorithm 1; and Step 8 searches a row for the qualified composition. In the worst case, the time taken by Step 8 is O(p). Hence, the worst-case time complexity of Algorithm 1 is O($p^2 m_{MAX}$).

The composition table has m_{MAX} rows. A composition in the table is located by a pair (r, l), which is denoted by $cps(r, l)$, where r is the row number in which the composition lies and l is the position subscript of the first job of the composition in the row. An example composition table is shown in Table 1.

4.2. Searching the composition table

The procedure of allocating jobs to a resource space with size r in a cluster proceeds as follows (Algorithm 3). First, it searches the composition table from the r-th row up to the first row to obtain the first row which is not null. Then, in this row, the algorithm selects the composition in which the number of jobs having been allocated is the least. If no jobs in the composition have been allocated, then these jobs are allocated to the resource space. If a job, J_i, whose size is s_i, in the composition has been allocated, a function (Algorithm 2) is called to search the s_i-th row for alternative jobs for J_i. The function is called recursively if a composition consisting of unallocated jobs cannot be found in the s_i-th row. The recursive call exits when there is only one composition in a searched row (i.e. no alternative jobs can be found) or when the composition is found in which no job is allocated.

If the algorithm fails to allocate jobs to the resource space with size r, it continues by trying to identify jobs to allocate to the resource space with size r=r-1. The searching procedure continues until r reaches 0, which means that no jobs can be allocated to the given resource space.

After the jobs which are to be allocated to a resource space have been determined, the schedule of these jobs can also be calculated (Step 11 in Algorithm 3).

It can be seen that Algorithm 3 always attempts to identify the jobs that maximally occupy the given resource space. By doing so, the number of resources left idle is minimized and the jobs are packed tightly.

The time complexity of Algorithm 3 (including Algorithm 2) is based on the number of jobs that are allocated. The best-case time complexity is O(1) while the worst-case time complexity is $O(p^2 n_{MAX})$, involving searching the whole composition table.

Algorithm 2. Calculating the alternative jobs for the allocated jobs in a composition *cps*(*r, l*)

Input: the position of the composition in the composition table (*r*, *l*); an array, *q* (used to contain alternative jobs).

Output: if succeeds, return 1; otherwise, return 0; (array *q* contains partial alternative jobs).

1. *lc*←*l*; *noway*←0;
2. **while** *lc* does not reach the end of the composition and *noway* equals 0
3. Get the job J_i pointed to by *lc*;
4. **if** J_i has not been allocated
5. Append J_i to array *q*;
6. **else if** there is only one composition in the s_i-th row of the composition table (s_i is the size of J_i), which consists of J_i itself
7. *noway*←1;
8. **else**
9. In the s_i-th row (except the composition consisting of J_i itself), get such a composition *cps*(s_i, *l*) in which the number of allocated jobs is minimum (if more than one composition shares the same minimum, select the first found);
10. **if** the number of allocated jobs in *cps*(s_i, *l*) is 0
11. Append the jobs in the composition *cps*(s_i, *l*) to array *q*;
12. **else**
13. Call Algorithm 2 with (s_i, *l*) and array *q* as the inputs;
14. **if** its output is 0
15. *noway*←1;
16. *lc*←*lc*+1;
17. **end while**;
18. **if** *noway* equals 0
19. **return** 1;
20. **else**
21. **return** 0;

Algorithm 3. Allocating jobs to a resource space (*t, r, cp*) in cluster C_i, where *t* is the time when the space is available, *r* is the size of the space and *cp* is the resource number that the space starts from

Input: the resource space (*t*, *r*, *cp*); an array, *q* (used to contain jobs allocated to the resource space).

1. *rc*←*r*;
2. **while** the *rc*-th row is NULL
3. *rc*←*rc*−1;
4. Get such a composition of jobs in which the number of allocated jobs is minimum in the *rc*-th row of the composition table (if more than one composition shares the same minimum, select the first found; suppose the composition found is *cps*(r_0, *l*));
5. **if** the number of allocated jobs in the composition is 0
6. Put the jobs in the composition in array *q*;
7. **else**
8. Call Algorithm 2 with (r_0, *l*) and array *q* as inputs;
9. **if** Algorithm 3 returns 0
10. *rc*←*rc*−1; Go to Step 2;
11. The starting time of these jobs is *t*; these jobs are allocated to the resources in the order in which they appear in array *q*, starting from resource *cp* (The pre-scheduling of these jobs forms part of the seed schedule sent to the TITANs at the local clusters).

4.3 Balance the load

In each local cluster, TITAN uses a genetic algorithm to adjust the seed schedule sent by MUSCLE, aiming to further improve the CP. Although MUSCLE has no control over the detailed operations of the genetic algorithm, it analyses the objective factors influencing the performance and allocates different levels of workload to each cluster through a heuristic so that the CP performance achieved by each cluster is well balanced.

The fundamental attribute of a cluster is the number of resources (the attribute of the service rates of resources is reflected in the execution times of jobs).

The fundamental attributes of the workload allocated to a cluster include:

- p_i: the number of jobs;
- $etSum_i$: the sum of execution times of all jobs;
- $sizeSum_i$: total job sizes;
- $slkSum_i$: total slack (the slack of a job is its deadline minus its execution time and its arrival time, which relates to the QoS of jobs).

The scheduling performance achieved in a cluster is determined by the resultant forces of these fundamental attributes. In addition, the packing potential for the parallel jobs allocated to a cluster is also considered a critical factor. This problem however is solved by Algorithm 3.

When the value of the attributes p_i, $etSum_i$ and $sizeSum_i$ in a cluster is lower, the cluster can be allocated more jobs than others, while the relation is opposite for $slkSum_i$ and m_i. Based on this relationship, these attributes are integrated to form a metric (ε) shown in Eq.2. When multiple clusters offer available resource space, the cluster with the smallest ε will be allocated the jobs. When more than one cluster has the same value of ε, the cluster

of the greatest size is given the highest priority. In the case of equal sized clusters, one is selected randomly.

$$\varepsilon = \frac{p_i \times etSum_i \times sizeSum_i}{slkSum_i \times m_i} \quad (2)$$

The complete scheduling procedure of MUSCLE is outlined in Algorithm 4. The time of Algorithm 4 is mainly dominated by Step 3 and Step 8. Their time complexities have been analysed in subsections 4.1 and 4.2.

Algorithm 4. MUSCLE scheduling

1. **while** the expected makespans of the jobs yet to be scheduled in all clusters (provided by TITAN) are greater than a predefined threshold
2. Collect the arriving jobs in the multicluster;
3. Call Algorithm 1 to construct the composition table for the collected jobs;
4. **do**
5. **for** each cluster **do**
6. Calculate ε using Eq.2;
7. Get the earliest available resource space in cluster C_i which has the minimal ε;
8. Call Algorithm 3 to allocate jobs to this space;
9. Update the earliest available resource space and the attribute values of the workload in C_i;
10. **while** all collected jobs have not been allocated;
11. Go to Step 1;

5. Experimental studies

A simulator has been developed to evaluate the performance of the scheduling mechanism in MUSCLE. The presented experimental results focus on demonstrating the performance advantages of the scheduling mechanism in MUSCLE over the scheduling policies frequently used in distributed systems. Weighted Random (WRAND) and Dynamic Least Load (DLL) policies are two selected representatives. WRAND only considers the processing capability of resources, while DLL considers both the workload level and the resource capability. MUSCLE takes additional factors into account, including the packing potential and the QoS requirements of parallel jobs.

In the experiments, the generated parallel jobs are submitted to the multicluster. MUSCLE, DLL or WRAND are used as the multicluster-level scheduling policies respectively, while in each case TITAN is used as the local-level scheduler in each local cluster. The combination of the weights for the over-deadline, the makespan and the idle-time is denoted by (W^o, W^m, W^i).

20,000 parallel jobs are generated; the submission of parallel jobs follows a Poisson process. The execution times of the jobs in cluster C_1 follow a bounded Pareto distribution [14] (shown in Eq.3), where e_l and e_u are the lower and upper limit of the execution time x.

$$f(x) = \frac{\alpha {e_l}^{\alpha}}{1-(e_l/e_u)^{\alpha}} x^{-\alpha-1} \quad (3)$$

If a job is scheduled to another cluster consisting of resources with a service rate sr_j, the execution time is determined by multiplying the ratio between sr_1 and sr_j (i.e., sr_1/sr_j). A job's deadline d_i is determined by Eq.4, where dr is the deadline ratio. dr follows a uniform distribution in [MIN_DR, MAX_DR]. The range is used to measure the deadline range.

$$d_i = \max\{e_{ij}\} \times (1+dr) \quad (4)$$

The job size can follow different probability distributions according to different application scenarios. [15] studies jobs whose sizes follow uniform or normal distribution, while [5] shows that the probability that a job uses fewer than n processors is roughly proportional to log$n$. These experiments are conducted under these different distributions to verify the effectiveness of the scheduling mechanism proposed in this paper. The results show similar patterns and hence, only the results for a certain representative distribution are presented in each experiment in this section.

Dynamic Least-Load (DLL) is a scheduling policy extensively used in heterogeneous systems [7][14]. The workload in cluster C_i, denoted as L_i, is computed using Eq.5. When a parallel job is submitted to the multicluster, DLL schedules the job to the cluster whose size is greater than the job's size and whose workload is the least.

$$L_i = \frac{\sum_{J_j \in WQ_i} e_{ji} \times s_j}{m_i u_i} \quad WQ_i \text{ is the set of jobs in } C_i \quad (5)$$

The Weighted Random (WRAND) policy is another frequently used scheduling policy in distributed systems [7][14]. When a job arrives at the multicluster, the WRAND policy first picks out all clusters whose sizes are greater than the job size (suppose these clusters are C_{i1}, C_{i2}, …, C_{ij}), and then schedules the job to a cluster C_{ik} ($1 \le k \le j$) with the probability proportional to its processing capability ($m_{ik}u_{ik}$).

The performance metrics evaluated in the experiments are the *mean comprehensive performance* (MCP) and the *performance balance factor* (PB). MUSCLE sends seed schedules to TITANs in the individual clusters C_i ($1 \le i \le n$), these TITANs are used to further improve the performance. When the performance in each generation of schedule stabilizes, the CP performance for cluster C_i, denoted by CP_i, is recorded. The MCP represents the average CP_i ($1 \le i \le n$), calculated by Eq.6, where p_i is the number of jobs allocated to C_i. The procedure continues until all generated jobs are processed. Each point in the performance curve is plotted as the average of the MCP calculated at each step. The PB is defined as the standard deviation of CP_i, shown in Eq.7.

$$MCP = \sum_{i=1}^{n} CP_i \times \frac{p_i}{p} \quad (6)$$

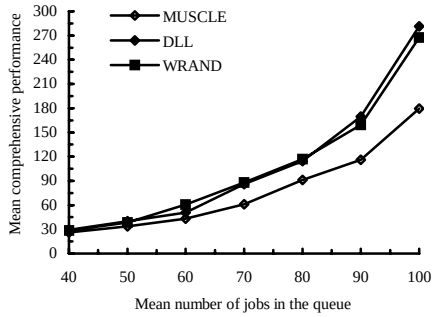
$$PB = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(CP_i - MCP)^2} \quad (7)$$

5.1. Workload levels

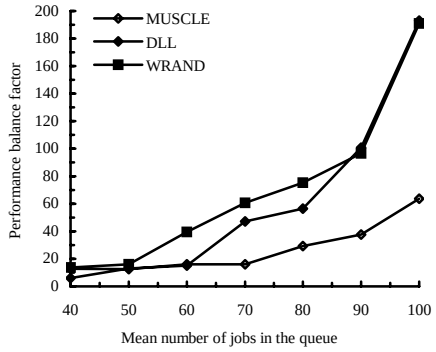
Fig.2.a and Fig.2.b demonstrate the performance difference among MUSCLE, DLL and WRAND policies under different workload levels. The workload level is measured by the mean number of jobs in the queue (for accommodating the collected jobs) in the multicluster.

Table 2. **The Multicluster setting in Fig.2**

Clusters	C_1	C_2	C_3	C_4
Size	20	16	12	10
Service rate ratio	1.0	1.2	1.4	1.6



(a)



(b)

Figure 2. **Performance comparison of MUSCLE, DLL and WRAND policies under different workload levels in terms of (a) mean comprehensive performance (MCP), and (b) performance balance factor (PB); The multicluster setting is in Table 2; (W_o, W_m, W_i)=(4, 3, 1); e_l/e_u=5/100; MIN_S/MAX_S=1/10; MIN_DR/MAX_DR=0/5**

It can be observed from Fig.2.a that MUSCLE outperforms the DLL and WRAND policies under all workload levels. This is because the jobs are packed tightly in the seed schedules sent by MUSCLE to the individual clusters. Therefore, further improvement by the genetic algorithm in each cluster is based on an excellent "seed". However, the jobs sent by DLL or WRAND to each cluster are random in terms of whether they can be packed tightly in the cluster or not. This reduces the possibility of achieving a high MCP in the multicluster. A further observation from Fig.2.a is that the benefits of MUSCLE over the other policies become increasingly pronounced as the workload increases. For example, when the mean number of jobs in the queue is 40, MUSCLE outperforms DLL by 12.1% in terms of the MCP. When the mean number of jobs is 100, the performance advantages increase to 56.7%. This is because when the number of jobs in the queue increases, MUSCLE can gather more job information and as a result make better allocation decisions among clusters.

As can be observed from Fig.2.b, when the mean number of jobs in the queue is less than 60, the PB performance achieved by MUSCLE is slightly worse than that of DLL. However, MUSCLE significantly outperforms DLL in all other cases. This can be explained as follows. When the workload is low, a small number of jobs miss their deadlines and the MCP is mainly caused by makespan and idle time. Therefore, DLL shows a more balanced MCP performance (though this does not mean DLL can achieve higher overall MCP performance). However, as the workload increases further, more jobs miss their deadlines. DLL ignores the QoS demands of these jobs. In contrast, MUSCLE takes the QoS demands into account so that the MCP performance remains balanced among the clusters.

5.2 Domain-level performance requirements

In the multicluster, different clusters can use different weight combinations to cater for their performance goals. Fig.3 compares the performance of MUSCLE, DLL and WRAND when the clusters in the multicluster utilise different weights to calculate the MCP, shown in Table 3. Fig.3.a, Fig.3.b and Fig.3.c show the performance comparison in terms of MCP while Fig.3.d, Fig.3.e and Fig.3.f show the comparison in terms of PB. The experiments are conducted under low, medium and heavy workloads (the mean number of jobs in the queue are 40, 100 and 160, respectively).

As can be observed from Fig.3.a, Fig.3.b and Fig.3.c, the performance of MUSCLE is superior to other policies in all cases in terms of the MCP. Further, the advantage of MUSCLE over DLL and WRAND is different under different workload levels. When the mean number of jobs in the queue is 40, the advantage of MUSCLE over DLL is 15.6%, while the advantage increases to 57.5% when the number of jobs is 100. When the workload increases further and the mean number of jobs in the queue is 160, the advantage of MUSCLE over DLL decreases to 13.7%. This is because when the workload level is low, the potential of MUSCLE cannot be fully utilised, while the advantage of MUSCLE becomes increasingly prominent as the workload level increases. However, the advantage diminishes when the workload level increases further and becomes saturated relative to the deadlines.

It can be observed from Fig.5.a, Fig.5.b and Fig.5.c that when the mean number of jobs in the queue is 40, the performance of MUSCLE in terms of PB is worse than that of DLL (MUSCLE still outperforms DLL in terms of the MCP). However under other workloads, MUSCLE performs significantly better than DLL and WRAND.

Table 3. **The weight combinations used in Fig.3**

	C_1	C_2	C_3	C_4
(W_o, W_m, W_i)	(12, 3, 1)	(8, 3, 1)	(4, 3, 1)	(1, 3, 1)

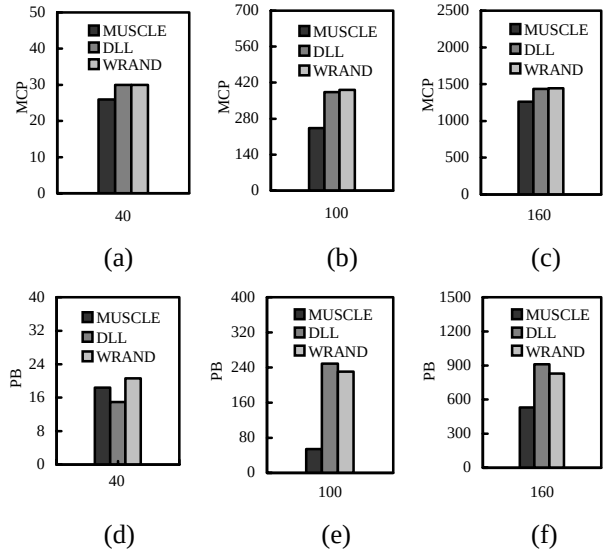


Figure 3. **Performance comparison of MUSCLE, DLL and WRAND in terms of MCP and PB; the *x*-axis is labelled by the mean number of jobs in the queue; e_l/e_u=5/100; MIN_S/MAX_S=1/10; MIN_DR/MAX_DR=0/5; the multicluster setting is the same as that found in Table 2**

5.3 Multicluster size

Fig.4 compares the performance of MUSCLE, DLL and WRAND under different multicluster sizes. In this experiment we use a cluster pool of size 10, where cluster sizes vary from 20 to 56 with increments of 4 and the service rates of all resources are equivalent. Initially, jobs are processed in the multicluster consisting of 4 clusters of the smallest size (20-32); then two clusters of greater size are added to the multicluster until all 10 clusters are used. The workload level is also increased accordingly, so that the ratio of the mean number of jobs in the queue to the total processing capability remains unchanged. In Fig.4, the initial mean number of jobs in the queue is 80. The probability that a job requests fewer than m_{MAX} resources is proportional to logm_{MAX} (m_{MAX} is the size of the largest cluster in the current multicluster).

As can be observed from Fig.4, MUSCLE outperforms DLL and WRAND in terms of MCP and PB in all cases. A further observation is that the advantage of MUSCLE over WRAND becomes more pronounced as the cluster size increases, while the trend is not obvious for the case of DLL. This can be explained as follows. WRAND allocates a fixed proportion of workload to each cluster. As the number of clusters increases, this policy becomes increasingly incompetent. DLL considers both workload and processing capability of each cluster. When the jobs sent by DLL to a cluster happen to have a good packing potential and their deadlines are not (on average) urgent, comparatively high performance can be achieved. However, DLL does not provide such a general scheme as that implemented in MUSCLE, which takes into account the packing potential and QoS requirements of jobs.

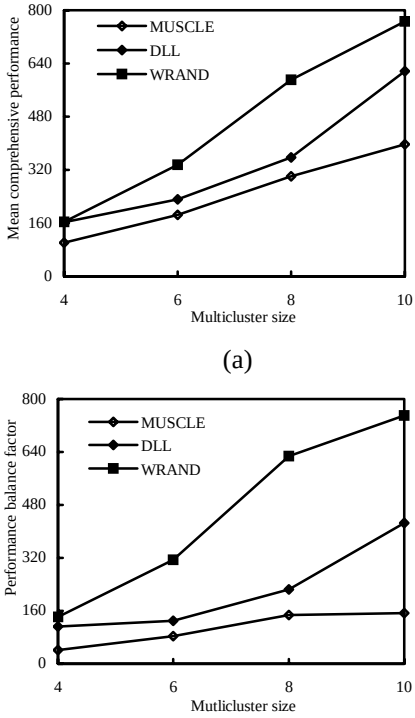
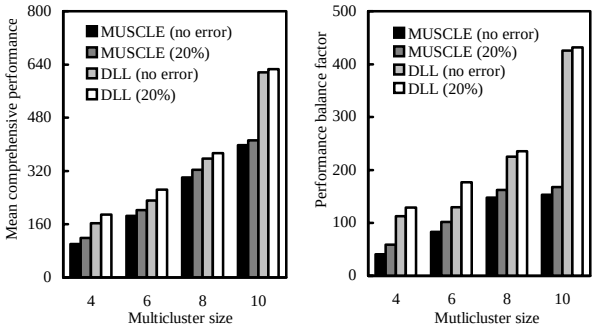


Figure 4. **Performance comparison as the multicluster size changes; (W_o, W_m, W_i)=(8, 3, 1); e_l/e_u=5/100; MIN_S/MAX_S=1/m_{MAX} (proportional to logn_{MAX}); MIN_DR/MAX_DR=0/5; the initial mean number of jobs in the queue is 80 and the cluster sizes vary in [20, 56]**

5.4 Prediction accuracy

In this work, a performance prediction toolkit (called PACE [11]) is used to predict the execution time of submitted jobs. PACE tends to overestimate a job's execution time [11][13]. Fig.5 demonstrates the effect of prediction accuracy on the scheduling performance, where the jobs are overestimated randomly by 0%-20% (it is rare that the

prediction error of PACE is greater than 20%). The prediction error has no impact on the WRAND policy since it only considers the processing ability of resources. The multicluster setting and the workload levels are the same as those in Fig.5.



(a) (b)

Figure 5. **Effect of the prediction error; the workload and multicluster setting are the same as those in Fig.4**

A general observation from Fig.5.a and Fig.5.b is that under both MUSCLE and DLL, the MCP and PB performance decreases when prediction errors exist. Further observations show that the performance deterioration is moderate. In the worst case (the multicluster size is 4), the MCP performance achieved by MUSCLE and DLL deteriorates by 14.9% and 13.7%, respectively; while by 3.5% and 1.5% in the best case (when the multicluster size is 10). This result suggests that the presence of prediction errors does not dramatically weaken the performance advantage of MUSCLE over DLL and WRAND.

6. Conclusions

A multicluster-level scheduler, called MUSCLE, is presented in this paper for the scheduling of parallel jobs with QoS demands in multiclusters and grids. Three metrics (over-deadline, makespan and idle-time) are combined with variable weights to evaluate the scheduling performance. MUSCLE is able to allocate jobs with high packing potential to the same cluster and further utilizes a heuristic to control the workload distribution among the clusters. Extensive experimental studies have been conducted to verify the performance advantages of MUSCLE. The results show that compared with traditional scheduling policies in distributed systems, the comprehensive performance (in terms of over-deadline, makespan and idle-time) is significantly improved and the jobs are well balanced across the multicluster.

7. References

[1] M. Barreto, R. Avila, and P. Navaux, "The MultiCluster model to the integrated use of multiple workstation clusters," *Proc. of the 3rd Workshop on Personal Computerbased Networks of Workstations*, 2000, pp. 71–80.

[2] R. Buyya, M. Murshed, D. Abramson, "A deadline and budget constrained cost-time optimization algorithm for scheduling task farming applications on global Grids," *In 2002 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA'02), Las Vegas, 2002.

[3] J. Cao, D. J. Kerbyson, and G. R. Nudd, "Performance Evaluation of an Agent-Based Resource Management Infrastructure for Grid Computing," *Proceedings of 1st IEEE/ACM International Symposium on Cluster Computing and the Grid* (CCGrid'01), 2001.

[4] H. Casanova, G. Obertelli, Francine Berman and Rich Wolski, "The AppLeS parameter sweep template: User-level middleware for the Grid," *Proc. the Super Computing Conference (SC'2000)*, Texas, 2000.

[5] D. Feitelson, L. Rudolph, "Metrics and Benchmarking for Parallel Job Scheduling," *4th Workshop on Job Scheduling Strategies for Parallel Processing*, 1998

[6] L. He, S. A. Jarvis, D. Bacigalupo, D. P. Spooner, X. Chen and G. R. Nudd, "Queueing Network-based Optimisation Techniques for Workload Allocation in Clusters of Computers," *2004 IEEE International Conference on Services Computing* (*SCC 2004*), Shanghai, China, 2004

[7] L. He, S. A. Jarvis, D. P. Spooner, G. R. Nudd, "Optimising static workload allocation in multiclusters," *18th IEEE International Parallel and Distributed Processing Symposium* (IPDPS'04), April 26-30, 2004, Santa Fe, New Mexico

[8] L. He, S. A. Jarvis, D. P. Spooner, G. R. Nudd, "Dynamic Scheduling of Parallel Real-time Jobs by Modelling Spare Capabilities in Heterogeneous Clusters," *Proceedings of IEEE International Conference on Cluster Computing* (*Cluster03*), pp. 2-10, Hong Kong, 2003

[9] X. He, X. Sun, and G. Laszewski, "QoS Guided Min-Min Heuristic for Grid Task Scheduling," *Journal of Computer Science and Technology, Special Issue on Grid Computing*, 2003.

[10] B. G. Lawson and E. Smirni, "Multiple-queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems," In the 8th *Job Scheduling Strategies for Parallel Processing*, 2002.

[11] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, J. S. Harper, S. C. Perry and D. V. Wilcox, "PACE: A Toolset for the Performance Prediction of Parallel and Distributed Systems," In *International Journal of High Performance Computing*, 1999.

[12] E. Shmueli and D. G. Feitelson, "Backfilling with lookahead to optimize the performance of parallel job scheduling," In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.), pp. 228-251, Springer-Verlag, 2003.

[13] D. P Spooner, S. A Jarvis, J Cao, S Saini, G. R Nudd, "Local Grid Scheduling Techniques using Performance Prediction," *IEE Proc. Comp. Digit. Tech.*, 150(2):87-96, 2003.

[14] X.Y. Tang, S.T. Chanson, "Optimizing static job scheduling in a network of heterogeneous computers," *29th Intl Conference on Parallel Processing*, 2000.

[15] C. Yu and C. R. Das, "Limit Allocation: An Efficient Processor Management Scheme for Hypercubes," *Proc. Int. Conf. Parallel Processing* (*ICPP94*), 1994.