Future Generation Computer Systems 29 (2012) 170–181

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Task granularity policies for deploying bag-of-task applications on global grids

Nithiapidary Muthuvelu [a,*], Christian Vecchiola [b], Ian Chai [a], Eswaran Chikkannan [a], Rajkumar Buyya [b]

[a] Multimedia University, Persiaran Multimedia, 63100 Cyberjaya, Selangor, Malaysia

[b] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computer Science and Software Engineering, The University of Melbourne, Carlton, Victoria 3053, Australia

ARTICLE INFO

Article history:
Received 23 July 2010
Received in revised form
20 March 2012
Accepted 22 March 2012
Available online 5 April 2012

Keywords:
Grid computing
Meta-scheduler
Lightweight task
Task granularity
Task group deployment

ABSTRACT

Deploying lightweight tasks individually on grid resources would lead to a situation where communication overhead dominates the overall application processing time. The communication overhead can be reduced if we group the lightweight tasks at the meta-scheduler before the deployment. However, there is a necessity to limit the number of tasks in a group in order to utilise the resources and the interconnecting network in an optimal manner. In this paper, we propose policies and approaches to decide the granularity of a task group that obeys the task processing requirements and resource-network utilisation constraints while satisfying the user's QoS requirements. Experiments on bag-of-task applications reveal that the proposed policies and approaches lead towards an economical and efficient way of grid utilisation.

1. Introduction

Grid computing [1–3] connects geographically distributed heterogeneous resources, forming a platform to run resource-intensive applications. A grid application contains a large number of tasks [4,5]; a meta-scheduler transmits each task file to a grid resource for execution and retrieves the processed task from the resource. The overall processing time of a task includes task invocation at the meta-scheduler, scheduling time, task file transmission to a resource, waiting time at the resource's local job queue, task execution time, and output file transmission to the meta-scheduler.

A lightweight or fine-grain task requires minimal execution time (e.g. less than one minute). Executing a large number of lightweight tasks one-by-one on a grid would result in a low computation–communication ratio as the total communication time will be high due to the overhead involved in handling each small-scale task [6]; the term *computation* refers to the task execution time, whereas *communication* refers to the task and output file transmission time. This issue can be explained from two point of views.

- The communication overhead increases proportionally with the number of tasks.
- The processing capability of a resource and the capacity of an interconnecting network will not be optimally utilised when dealing with lightweight tasks. For example:
  - assume that a high-speed machine allows a user to utilise its CPU for x seconds. Executing lightweight tasks one at a time on the machine will miss the full processing speed (e.g. x^* Million Instructions per Second) of the machine within x seconds due to the overhead involved in invoking and executing each task;
  - transmitting task and output files one-by-one to and from a resource may underutilise the achievable bandwidth if the files are very small.

Hence, deploying lightweight tasks on a grid would lead to inefficient resource and network utilisation, resulting in an unfavourable application throughput. This statement is proven with experiments in Section 5.3.1 of this paper. The experiments show that grouping the lightweight tasks before the deployment increases resource utilisation and reduces the overall application processing time significantly. This stimulates the need for optimal *task granularity* (the number of tasks that should be grouped in a batch) for each resource at runtime.

In this paper, we present the factors that highly affect the decision on task granularity which result in a set of policies for determining task granularities at runtime. The policies are then incorporated in the scheduling strategies of a meta-scheduler to be tested in a grid environment. Our goal is to reduce the overall application processing time while maximising the usage of

* Corresponding author. Tel.: +60 383125429; fax: +60 383125264.
E-mail addresses: nithiapidary@mmu.edu.my, m.nithia@gmail.com (N. Muthuvelu).

0167-739X/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2012.03.022

resource and network capacities, and obeying the quality of service (QoS) requirements.

The scheduling strategies are designed to handle computation-intensive, parametric and non-parametric sweep applications. They assume that all the tasks in an application are independent, computational, and have a similar compilation platform.

The rest of the paper is organised as follows: The related work is explained in Section 2. Section 3 describes the factors involved in deciding the task granularity followed by the task granularity policies. Section 4 presents the approaches to deal with the issues induced by the task granularity policies. The process flow of the proposed meta-scheduler is explained in a subsection of Section 4. Section 5 brings the performance analysis of the scheduling strategies. Finally, Section 6 concludes the paper by suggesting future work.

2. Related work

Task granularity adaptation has been one of the most important research problems in batch processing.

Algorithms pertaining to sequencing tasks in multiple batches for executions on a single machine to minimise the processing delays were demonstrated in [7,8]. Following from these experiments, Mosheiov and Oron proposed an additional parameter, maximum/minimum batch size, to control the number of tasks to be grouped in a batch in [9].

James et al. [10] scheduled equal numbers of independent jobs using various scheduling algorithms to a cluster of nodes. However, their attempt caused an additional overhead as the nodes were required to be synchronised after each job group execution iteration.

Sodan et al. [11] conducted simulations to determine the optimal number of jobs in a batch to be executed in a parallel environment. The total number of jobs in a batch is optimised based on minimum and maximum group size, average run-time of the jobs, machine size, number of running jobs in the machine, and minimum and maximum node utilisation. These simulations did not consider varying network usage or bottlenecks. In addition, the total number of jobs in a batch is constrained with static upper and lower bounds.

Work towards adapting computational applications to constantly changing resource availability has been conducted by Maghraoui et al. [12]. A specific API with special constructs is used to indicate the atomic computational units in each user job. Upon resource unavailability, the jobs are resized (split or merged) before being migrated to another resource. The special constructs in a job file indicates the split or merge points.

A few simulations have been conducted to realise the effect of task grouping in a grid [13]. The tasks were grouped based on resource's Million Instructions Per Second (MIPS) and task's Million Instructions (MI). MIPS or MI are not the preferred benchmark matrices as the execution times for two programs of similar MI but with different compilation platforms can differ [14]. Moreover, a resource's full processing capacity may not be available all the time because of I/O interrupt signals.

In 2008 [15], we designed a scheduling algorithm that determines the task granularity based on QoS requirements, task file size, estimated task CPU time, and resource constraints on maximum allowed CPU time, maximum allowed wall-clock time, maximum task file transmission time, and task processing cost per time unit. The simulation shows that the scheduling algorithm performs better than conventional task scheduling by 20.05% in terms of overall application processing time when processing 500 tasks. However, it was assumed that the task file size is similar to the task length which is an oversimplification as the tasks may contain massive computation loops.

In our previous work [16], we enhanced our scheduling algorithm by treating the file size of a task separately from its processing needs. The algorithm also considers two additional constraints: space availability at the resource and output file transmission time. In addition, it is designed to handle unlimited number of user tasks arriving at the scheduler at runtime.

This paper is an improvement of our previous work [16,17]. We enhanced our scheduling strategies to coordinate with cluster-based resources without synchronisation overhead. The strategies support the tasks from both parametric and non-parametric sweep applications. We developed a meta-scheduler, implemented our proposed task grouping policies and approaches, and tested the performance in a real grid environment. The user tasks are transparent to the meta-scheduler and there is no need for a specific API to generate the tasks.

3. Factors influencing the task granularity

Table 1 depicts the terms or notations and the corresponding definitions that will be used throughout this paper.

Our aim is to group multiple fine-grain tasks into a batch before deploying the batch on a resource. When adding a task into a batch or a group, the processing need of the batch will increase. This demands us to control the number of tasks in a batch or the resulting granularity. As a grid resides in a dynamic environment, the following factors affect the task granularity for a particular resource:

- The processing requirements of the tasks in a grid application.
- The processing speed and overhead of the grid resources.
- The resource utilisation constraints imposed by the providers to control the resource usage [18].
- The bandwidths of the interconnecting networks [19].
- The QoS requirements of an application [20].

Fig. 1 depicts the information flow pertaining to the above-mentioned factors in a grid environment. The grid model contains three entities: User Application; Meta-Scheduler; and Grid Resources. (1) The first input set to the meta-scheduler comes from the user application which contains a bag of tasks (BoT).

- *Tasks*: A task (T) contains files relevant to the execution instruction, library, task or program, and input data.
- *Task Requirements*: Each task is associated with task requirements or characteristics which consist of the size of the task file (*TFSize*), the estimated size of the output file (*OFSize*), and the estimated CPU time of the task (*ETCPUTime*). The *ETCPUTime* is an estimation given by the user based on sample task executions on the user's local machine. In our context, the *ETCPUTime* of a task is measured at the application or task level (not at processor level): from the start of a task execution till the end of the task execution.
- *QoS*: The user budget (*UBudget*) and deadline (*UDeadline*) allocated for executing all the tasks in the BoT.

(2) The second input set to the meta-scheduler is from the grid resources (*GR*) participating in the environment. The resource providers impose utilisation constraints on the resources in order to avoid the resources from being overloaded or misused [18,21]. The utilisation constraints of a particular resource, R, are:

- *Maximum Allowed CPU Time (MaxCPUTime)*: The maximum time allowed for the execution of a task or a batch at a resource.
- *Maximum Allowed Wall-Clock Time (MaxWCTime)*: The maximum time a task or a batch can spend at the resource. This encompasses task CPU time and task processing overhead at the resource (task waiting time, and task packing and unpacking overhead).

Table 1
Terms and definitions.

User application and task requirements	
BoT	**B**ag **o**f **T**asks
T	A **T**ask
TFSize	**T**ask **F**ile **S**ize
ETCPUTime	**E**stimated **T**ask **CPU T**ime
OFSize	**O**utput **F**ile **S**ize
UBudget	**U**ser **B**udget allocated for a grid application
UDeadline	**U**ser **D**eadline for completing a grid application
Grid resources and resource-network utilisation constraints	
GR	A set of **G**rid **R**esources
R	A grid **R**esource
MaxCPUTime	**M**aximum allowed **CPU T**ime
MaxWCTime	**M**aximum allowed **W**all-**C**lock **T**ime
MaxSpace	**M**aximum allowed storage **S**pace for the task and output files
PCost	Task **P**rocessing **C**ost per time unit
MaxTransTime	**M**aximum allowed file **T**ransmission **T**ime between the meta-scheduler and a resource
Task categories and the average requirements of each category	
$TFSize_{Cl}$	**T**ask **F**ile **S**ize **C**lass **I**nterval
$ETCPUTime_{Cl}$	**E**stimated **T**ask **CPU T**ime **C**lass **I**nterval
$OFSize_{Cl}$	**O**utput **F**ile **S**ize **C**lass **I**nterval
TCat	**T**ask **C**ategory
AvgTFSize	**A**verage **T**ask **F**ile **S**ize of a task category
AvgETCPUTime	**A**verage **E**stimated **T**ask **CPU T**ime of a task category
AvgOFSize	**A**verage **O**utput **F**ile **S**ize of a task category
Average deployment metrics of each task category–resource pair E.g. Average deployment metrics of a task category k on a resource j, $TCat_k - R_j$	
$AvgSTRTime_{k,j}$	**A**verage meta-**S**cheduler **T**o **R**esource task file transmission **T**ime
$AvgCPUTime_{k,j}$	**A**verage task **CPU T**ime
$AvgWCTime_{k,j}$	**A**verage task **W**all-**C**lock **T**ime
$AvgPCost_{k,j}$	**A**verage task **P**rocessing **C**ost
$AvgRTSTime_{k,j}$	**A**verage **R**esource **T**o meta-**S**cheduler output file transmission **T**ime
$AvgOverhead_{k,j}$	**A**verage task processing **O**verhead
$AvgTRTime_{k,j}$	**A**verage **T**ask **T**urnaround **T**ime

- *Maximum Allowed Storage Space (MaxSpace)*: Maximum space that a task or a batch (including the corresponding output files) can occupy at the resource at a time.
- *Task Processing Cost (PCost)*: The task execution cost per time unit charged by a resource.

(3) The third input is the network utilisation constraint:

- *Maximum Allowed File Transmission Time (MaxTransTime)*: The tolerance threshold or the maximum time that a meta-scheduler can wait for the task and output files to be transmitted to and from a resource.

In order to form a task group, having these input sets, the policies on task granularity can be generalised as follows.

Assuming that a bag of tasks contains n tasks,

$$BoT = \{T_0, T_1, T_2, T_3, T_4, \ldots, T_{n-1}\}; \quad BoT_{TOTAL} = n$$

and the grid environment consists of r resources,

$$GR = \{R_0, R_1, R_2, R_3, R_4, \ldots, R_{r-1}\}; \quad GR_{TOTAL} = r$$

the seven policies for determining the granularity of a task group, *TG*, for a grid resource, *Ri*, would be:

Policy 1: *TG* CPU time $\leq MaxCPUTime_{Ri}$
Policy 2: *TG* wall-clock time $\leq MaxWCTime_{Ri}$
Policy 3: *TG* and output transmission time $\leq MaxTransTime_{Ri}$
Policy 4: *TG* and output file size $\leq MaxSpace_{Ri}$
Policy 5: *TG* turnaround time $\leq$ Remaining *UDeadline*
Policy 6: *TG* processing cost $\leq$ Remaining *UBudget*
Policy 7: Number of tasks in *TG* $\leq$ Remaining BoT_{TOTAL}

where, Policies 1–4 are related to resource-network utilisation constraints, Policies 5–6 are on QoS requirements, and Policy 7 is to check the task availability.

However, there are three issues that affect the task granularity decision according to these seven policies as the grid resides in an environment of heterogeneous resources and fluctuating network conditions.

ISSUE I: A grid resource can be a supercomputer, a cluster of multiple nodes, a node with multiple processing cores, etc. The wall-clock time of a task is influenced by the speed of a resource's local job scheduler (e.g. SGE, PBS, LSF, Libra, thread-level schedulers, etc.) and the current processing load of the resource. In order to obey the Policies 2 and 5, one should know the overheads of resources' queueing systems in advance.

ISSUE II: Task CPU time differs according to the resources' processing capabilities. For example, a group of five tasks might be handled by *Resource A* smoothly, whereas it may exceed the maximum allowed CPU time or wall-clock time of *Resource B*, in spite of having a similar architecture as *Resource A*. In addition, the CPU time highly depends on the programming model and compilation platform of the task. Relevant to this issue is the term 'fine-grain' or 'lightweight' which is very fuzzy in nature. For example, an executable file with 20,000 instructions can be a fine-grain task for a machine that processes 10,000 instructions per second. However, a machine that processes 10 instructions per second will consider the executable file as an average- or coarse-grain task. Hence, we should learn the resource speed and the processing need of the tasks prior to the task grouping in order to obey Policies 1 and 6.

ISSUE III: Task grouping increases the file size to be transmitted to and from the resources. Hence, the network will be overloaded if there is no control over the number of files to be grouped into a single batch. Here, we should consider the varying achievable bandwidth and the latency of the interconnected network [19,22]. For example, the network bandwidth at time t_x may support the

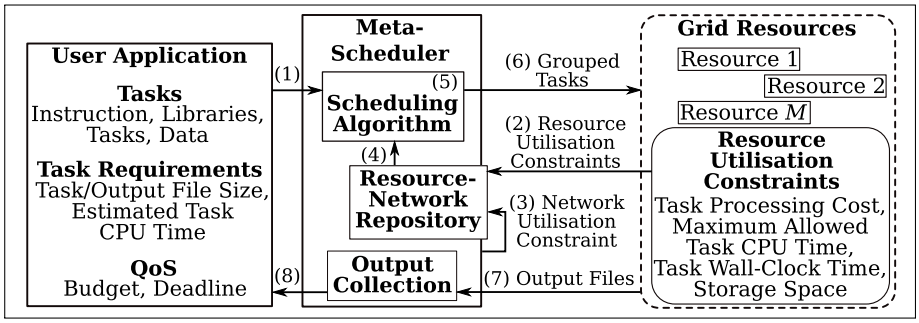


Fig. 1. The meta-scheduler and the information flow.

transmission of a batch of seven tasks, however, at time t_y this may result in a heavily-loaded network, leading to an unfavourable transmission time (where $x < y$). Therefore, we should determine the appropriate file size that can be transferred at a particular time in order to utilise the achievable bandwidth and obey the Policies 3 and 5.

These issues are caused by the dynamic nature of a grid environment. Hence, there is a need for periodic observations on the status of the grid.

4. Implementation of the meta-scheduler

In our meta-scheduler, the issues mentioned in Section 3 are tackled using three approaches in the following order:

- Task Categorisation: Categorising the tasks according to their requirements.
- Task Category–Resource Benchmarking: Learning the behaviour of the grid in response to the executions of the categorised tasks.
- Average Analysis: Learning the behaviour of the grid periodically in response to the executions of the categorised tasks.

These three approaches and the process flow of the meta-scheduler are explained in the following subsections.

4.1. Task categorisation

The tasks in a parametric-sweep BoT are similar in terms of task file size, while varying in terms of CPU time and output file size. The tasks in a non-parametric sweep BoT vary in terms of CPU time, and the task and output file size. When adding a task into a group, the *TFSize*, *ETCPUTime*, and *OFSize* of the task group get accumulated. Hence, the scheduler should select the most appropriate tasks from the BoT (in a timely manner) so that the resulting task group satisfies all the seven policies. This demands the need for proper task file management and file searching strategies.

Here, we arrange the tasks in a tree structure based on certain class interval thresholds applied to *TFSize*, *ETCPUTime*, and *OFSize*. This approach divides the tasks into categories according to the task file size class interval ($TFSize_{CI}$), followed by the estimated task CPU time class interval ($ETCPUTime_{CI}$), and then the output file size class interval ($OFSize_{CI}$).

Algorithm 1 depicts the level 1 categorisation in which the tasks are divided into categories (*TCat*) based on *TFSize* of each task and the $TFSize_{CI}$. The range of a category is set according to $TFSize_{CI}$. For example, the range of:

$TCat_0$: 0 to $(1.5 \times TFSize_{CI})$
$TCat_1$: $(1.5 \times TFSize_{CI})$ to $(2.5 \times TFSize_{CI})$
$TCat_2$: $(2.5 \times TFSize_{CI})$ to $(3.5 \times TFSize_{CI})$

Algorithm 1: Level 1 Task Categorisation.

```
Data: Requires TFSize of each T and TFSize_CI
1  for i ← 0 to BoT_TOTAL do
2      if T_{i−TFSize} < TFSize_CI then
3          TCatID ← 0
4      else
5          ModValue ← T_{i−TFSize} mod TFSize_CI
6          BaseValue ← T_{i−TFSize} − ModValue
7          if ModValue < TFSize_CI/2 then
8              TCatID ← (BaseValue/TFSize_CI) − 1
9          else
10             TCatID ← ((BaseValue + TFSize_CI)/TFSize_CI) − 1
11     T_i belongs to TCat of ID TCatID
```

The category ID (*TCatID*) of a task is 0 if its *TFSize* is less than the $TFSize_{CI}$ (line 2, 3). Otherwise, the mod and base values (line 5, 6) of the *TFSize* are computed to determine the suitable category range. For example, when $TFSize_{CI} = 10$ size unit, then

tasks with $(0 < TFSize < 15)$ belong to $TCat_0$
tasks with $(15 \leq TFSize < 25)$ belong to $TCat_1$
tasks with $(25 \leq TFSize < 35)$ belong to $TCat_2$

This is followed by level 2 categorisation in which the categories from level 1 are further divided into sub-categories according to *ETCPUTime* of each task and $ETCPUTime_{CI}$. A similar categorisation algorithm is applied for this purpose. For example, when $ETCPUTime_{CI} = 6$ time unit, then the tasks in

$TCat_0$ with $(0 < ETCPUTime < 9)$ belong to $TCat_{0-0}$
$TCat_0$ with $(9 \leq ETCPUTime < 15)$ belong to $TCat_{0-1}$
$TCat_1$ with $(0 < ETCPUTime < 9)$ belong to $TCat_{1-0}$
$TCat_1$ with $(9 \leq ETCPUTime < 15)$ belong to $TCat_{1-1}$

Subsequently, level 3 categorisation divides the categories from level 2 into sub-categories based on *OFSize* and $OFSize_{CI}$. For example, when $OFSize_{CI} = 10$ size unit, then the tasks in

$TCat_{0-0}$ with $(0 < OFSize < 15)$ belong to $TCat_{0-0-0}$
$TCat_{0-0}$ with $(15 \leq OFSize < 25)$ belong to $TCat_{0-0-1}$
$TCat_{0-1}$ with $(0 < OFSize < 15)$ belong to $TCat_{0-1-0}$
$TCat_{0-1}$ with $(15 \leq OFSize < 25)$ belong to $TCat_{0-1-1}$

Fig. 2 presents an instance of task categorisation when $TFSize_{CI} = 10$, $ETCPUTime_{CI} = 6$, and $OFSize_{CI} = 10$. The categories at each level are created only when there is at least one task belonging to that particular category. For each resulting *TCat*, the average task requirements are computed, namely, average task file size (*AvgTFSize*), average estimated task CPU time (*AvgETCPUTime*), and average output file size (*AvgOFSize*). These average task requirements will be used by the meta-scheduler in a later process (in Section 4.2).

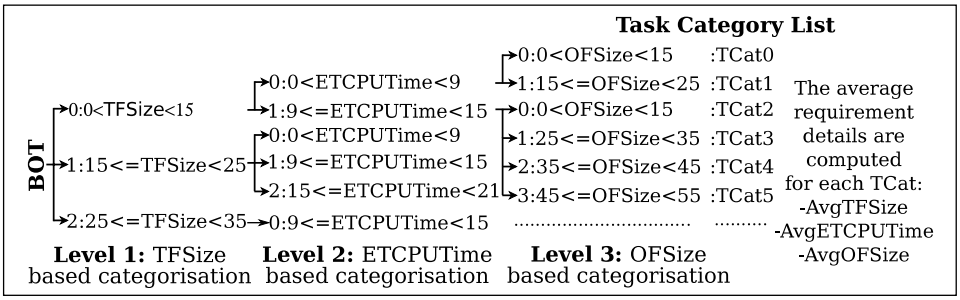


Fig. 2. Task categorisation.

This file organisation allows the meta-scheduler to easily locate the category that obeys the seven policies. Then, it selects a task file from the particular category to be added into a batch.

The order of the categorisation process can be altered; e.g. in level 1 categorisation, the tasks can be divided according to $ETCPUTime_{CI}$ instead of $TFSize_{CI}$. The resulting categories are not affected by the categorisation order, but merely depend on the class interval used at each level. Small class intervals can be used to increase the number of resulting categories in order to achieve better accuracy when selecting a task for a batch.

4.2. Task category–resource benchmarking

As mentioned in Section 3, the performance and overhead of the resources or the network cannot be estimated based on some written specifications. Hence, we suggest a benchmark phase where a few tasks are selected from the BoT and deployed on the resources. This is to study the behaviour of the grid when dealing with the user tasks before scheduling the entire BoT.

For this purpose, first, we determine the dominating categories based on the total number of tasks in the categories. Then, we select p tasks from the first 20%–50% of the dominating categories and send to each resource. The total number of benchmark tasks, $BTasks_{TOTAL}$, with m dominating categories can be expressed as:

$$BTasks_{TOTAL} = m \times p \times GR_{TOTAL}. \quad (1)$$

Upon retrieving the processed output files of a benchmark task, the remaining *UBudget* and *UDeadline* are updated accordingly. Then, the following seven deployment metrics of the task are computed:

task file transmission time (meta-scheduler to resource); CPU time; wall-clock time; processing cost; output file transmission time (resource to meta-scheduler); processing overhead (task waiting time, and task packing and unpacking time at the resource); and turnaround time.

Finally, after completing all the benchmark tasks, the average of each deployment metric is computed for each task category–resource pair. For a category k, the average deployment metrics on a resource j are expressed as average deployment metrics of $TCat_k - R_j$, which consist of:

average task file transmission time ($AvgSTRTime_{k,j}$); average CPU time ($AvgCPUTime_{k,j}$); average wall-clock time ($AvgWCTime_{k,j}$); average processing cost ($AvgPCost_{k,j}$); average output file transmission time ($AvgRTSTime_{k,j}$); average processing overhead ($AvgOverhead_{k,j}$); and average turnaround time ($AvgTRTime_{k,j}$).

It can be noticed that not all the categories are participating in this benchmark. Therefore, the average deployment metrics of those categories will be updated based on the average ratio of the categories participated in the benchmark. For example, assume that $TCat_0$, $TCat_1$, and $TCat_3$ have participated in the benchmark. The average CPU time of $TCat_2$ for a resource j can be updated as follows:

$AvgCPUTime_{2,j}$ based on $TCat_0$
$= AvgETCPUTime_2 \times (AvgCPUTime_{0,j}/AvgETCPUTime_0)$
$AvgCPUTime_{2,j}$ based on $TCat_1$
$= AvgETCPUTime_2 \times (AvgCPUTime_{1,j}/AvgETCPUTime_1)$
$AvgCPUTime_{2,j}$ based on $TCat_3$
$= AvgETCPUTime_2 \times (AvgCPUTime_{3,j}/AvgETCPUTime_3)$.

Assuming that m task categories have participated in the benchmark phase, then the $AvgCPUTime_{2,j}$ can be formulated as,

$$AvgCPUTime_{2,j} = \left(AvgETCPUTime_2 \times \sum_{k=0}^{m-1} (AvgCPUTime_{k,j}/AvgETCPUTime_k) \right) \Big/ \mathrm{m}$$

where k denotes the TCatID in the benchmark and k takes specific values in the range $\{0, 1, 2, \ldots, TCat_{TOTAL} - 1\}$.

A similar rule is applied to update the other task deployment metrics in the following order:

$$AvgSTRTime_{i,j} = \left(AvgTFSize_i \times \sum_{k=0}^{m-1} (AvgSTRTime_{k,j}/AvgTFSize_k) \right) \Big/ \mathrm{m}$$

$$AvgCPUTime_{i,j} = \left(AvgETCPUTime_i \times \sum_{k=0}^{m-1} (AvgCPUTime_{k,j}/AvgETCPUTime_k) \right) \Big/ \mathrm{m}$$

$$AvgRTSTime_{i,j} = \left(AvgOFSize_i \times \sum_{k=0}^{m-1} (AvgRTSTime_{k,j}/AvgOFSize_k) \right) \Big/ \mathrm{m}$$

$$AvgPCost_{i,j} = \left(AvgCPUTime_{i,j} \times \sum_{k=0}^{m-1} (AvgPCost_{k,j}/AvgCPUTime_{k,j}) \right) \Big/ \mathrm{m}$$

$$AvgOverhead_{i,j} = \left(\sum_{k=0}^{m-1} AvgOverhead_{k,j} \right) \Big/ \mathrm{m}$$

$$AvgWCTime_{i,j} = AvgCPUTime_{i,j} + AvgOverhead_{i,j}$$

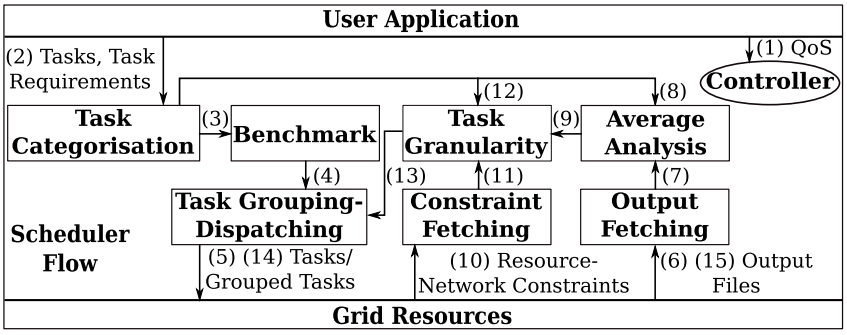


Fig. 3. Process flow of the meta-scheduler system.

$$AvgTRTime_{i,j} = AvgWCTime_{i,j} + AvgSTRTime_{i,j} + AvgRTSTime_{i,j}$$

where,

k denotes the TCatID in the benchmark and k takes specific values in the range $\{0, 1, 2, \ldots, TCat_{TOTAL} - 1\}$.

i denotes the TCatID missed the benchmark and i takes specific values in the range $\{0, 1, 2, \ldots, TCat_{TOTAL} - 1\}$.

$j = 0, 1, 2, \ldots, GR_{TOTAL} - 1$ (Grid resource ID).

$m =$ Total categories in the benchmark.

In short, this benchmark phase studies the response or performance of the grid resources and the interconnecting network on each task category.

4.3. Average analysis

Task categorisation and task category–resource benchmarking help the meta-scheduler to learn the behaviour of the grid. Now, we can group the tasks according to the seven policies as in Section 3.

However, as the grid operates in a dynamic environment, the deployment metrics of a task category may not reflect the latest grid environment after a time period [23]. Hence, the meta-scheduler should update the deployment metrics of each $TCat_k - R_j$ pair periodically based on the latest processed task groups.

A group or batch may contain tasks from various categories. The batch is accepted by a resource as a single task. Upon execution, the deployment metrics can only be computed for a batch rather than for the individual tasks in the batch. Hence, our meta-scheduler practices the followings steps to update the deployment metrics of each $TCat_k - R_j$ pair:

For a resource R_j,

1. Get the latest processed task groups and their 'actual' deployment metrics. Here, we select groups that were successfully processed by R_j within the last 10 min.
2. Identify the tasks and their categories in each group. Based on the previous $TCat_k - R_j$ average metrics, compute the 'estimated' task deployment metrics that each group should obtain.
3. For each group, compute the ratios 'estimated': 'actual' of the seven deployment metrics. Use these ratios to estimate and update the latest $TCat_k - R_j$ average details.
4. For those categories which did not participate in the latest processed task groups, update their $TCat_k - R_j$ average details based on the ratios of the participated categories as explained in Section 4.2.

This periodic average analysis on the latest processed task groups will keep updating the meta-scheduler with the current performance of each $TCat_k - R_j$ pair.

4.4. The meta-scheduler

We developed a meta-scheduler that practices the proposed task granularity policies and approaches to schedule and deploy fine-grain tasks on individual and cluster-based resources. The meta-scheduler is implemented in Java using multi-threading features. Fig. 3 presents the process flow of the entire meta-scheduler system.

There are eight modules involved in the system, namely, *Controller*, *Task Categorisation*, *Benchmark*, *Task Grouping-Dispatching*, *Output Fetching*, *Constraint Fetching*, *Average Analysis*, and *Task Granularity*.

(1) The *Controller* manages the flow of the meta-scheduler. It ensures that the QoS requirements are satisfied at runtime. (2) The *Task Categorisation* categorises the user tasks as mentioned in Section 4.1. (3) Then, it invokes the *Benchmark* which selects $BTasks_{TOTAL}$ benchmark tasks from the BoT which will be dispatched (4,5) to the grid resources by *Task Grouping-Dispatching*. (6) The *Output Fetching* then collects the processed benchmark tasks.

After the benchmark phase, (7,8) the *Average Analysis* studies the task category–resource average deployment metrics. (10) The *Constraint Fetching* keeps retrieving the resource-network utilisation constraints periodically, which will be used as conditions in task grouping policies.

(12) Having the categorised tasks, (9) $TCat_k - R_j$ average deployment metrics, and (11) the resource-network utilisation constraints, the *Task Granularity* determines the number of tasks from various categories that can be assigned into one batch for a particular resource. When selecting a task from a category, the estimated deployment metrics of the group are accumulated from the average deployment metrics of the particular task category. The resulting task granularity must satisfy all the seven policies mentioned in Section 3 of this paper.

The task categorisation process derives the need for enhancing Policy 7 to control the total number of tasks that can be selected from a category. Hence, the Policy 7 can be expressed as follows:

Policy 7: Total tasks in TG from a $TCat_k \leq size\_of\,(TCat_k)$

where,

$k = 0, 1, 2, \ldots, TCat_{TOTAL} - 1$ (TCatID).

(13,14) After setting the task granularity, the *Task Grouping-Dispatching* groups the selected tasks and transfers the group to the designated resource. Technically, the module compresses the selected tasks into one file. (15) The processed groups are then retrieved by the *Output Fetching*, and the remaining *UBudget* and *UDeadline* are updated accordingly. The cycle (10–15) continues for a certain time period and the *Controller* signals the *Average Analysis* to update the average deployment metrics of each $TCat_k - R_j$ based on the latest processed task groups. The subsequent decision on task granularity will be according to the updated average deployment metrics.

Table 2
Grid resources.

ID	Resource name (location)	Total nodes	Total cores	Operating system, speed, RAM
R_0	ibm.mygridusbio.net.my (MIMOS, Malaysia)	8	8 × 4	CentOS, 2.60 GHz, 15 GB
R_1	sun.mygridusbio.net.my (MIMOS, Malaysia)	16	16 × 8	CentOS, 2.20 GHz, 32 GB
R_2	belle.csse.unimelb.edu.au (UNIMELB, Australia)	1	1 × 4	Ubuntu, 2.80 GHz, 2 GB
R_3	sigs.mmu.edu.my (MMU, Malaysia)	1	1 × 4	OpenSUSE, 2.40 GHz, 2 GB

Eliminating synchronisation overhead: The resources are heterogeneous in terms of processing speed. When a particular resource completes its benchmark tasks, a set of instances of *Average Analysis*, *Constraint Fetching*, *Task Granularity*, and *Task Grouping-Dispatching* are initiated for that resource. Thus, the subsequent task scheduling, grouping, and deployment activities can be conducted for the resource without a delay. This eliminates the synchronisation overhead; there is no need for the meta-scheduler to wait for all the resources to complete their benchmark tasks before performing the next iteration of task grouping and deployment.

The grid resources are not synchronised throughout the scheduling process. A task group is dispatched to a resource when the resource becomes idle.

Scheduling tasks to cluster nodes: In our grid environment, there is no direct connection between the meta-scheduler and the cluster nodes. The meta-scheduler can only access the head or master node of the cluster. It then submits the tasks to the cluster job queueing system which assigns the tasks to the idle nodes in the cluster. Therefore, before planning the task deployment, there is a need to enquire the cluster job queueing system about the number of idle nodes. The meta-scheduler prepares sufficient number of task groups so that it can utilise all the idle nodes simultaneously.

When a processed task group is retrieved by the *Output Fetching*, the meta-scheduler plans the next task group to the cluster. Throughout the scheduling process, the meta-scheduler keeps track of the average number of nodes available in the cluster. This value will be used by the *Average Analysis* to measure the average performance of the cluster resource as a whole instead of the individual cluster nodes.

5. Performance evaluation

In this section, we conduct three phases of experiments using our meta-scheduler to realise the effects of the task granularity policies. The following subsections deliver detailed explanations on the experimental set-up and the performance analysis.

5.1. Grid environment

Table 2 presents the information of four grid resources used for our experiments. MIMOS, Malaysia [24] is an active site connected to the EGEE infrastructure [25]. For our experiments, we use two PBS clusters, namely R_0 and R_1, from MIMOS, R_2 from the University of Melbourne (UNIMELB), Australia, and R_3 from Multimedia University (MMU), Malaysia. The client machine which runs the meta-scheduler is located in MMU, the same domain as R_3. The machine is equipped with a dual-core CPU of 2.00 GHz and 3 GB RAM.

The client machine communicates with MIMOS and UNIMELB resources via conventional Internet connections, whereas it uses Intranet access to MMU resources. Simple SSH, SCP, and RSH protocols are used for authentications, file transmissions, and task executions.

5.2. Grid applications

The experiments are conducted using two types of BoT applications, namely, parametric and non-parametric sweep applications.

Application 1: *Non-parametric sweep application*: This BoT comprises instances of six computational programs, namely, heat distribution, linear equation, finite differential equation, and three versions of PI computation. The instances of each program are to be executed using various parameter sets. There are 634 tasks in this BoT and the requirements of the tasks are *TFSize* (7–10 KB), *ETCPUTime* (0.07–10 min), and *OFSize* (0.05–5950 KB).

The majority of the tasks are considered fine-grain: 79.02% of the tasks have *ETCPUTime* $\leq$ 2 min and 78.86% of the tasks have *OFSize* $\leq$ 1000 KB.

Application 2: *Parametric sweep application*: As a test case for the parameter sweep application, we use the Evolutionary Multi-Objective Optimiser (EMO) [26]. EMO is an evolutionary optimiser based on genetic algorithms [27] and it constitutes a perfect candidate to devise a parameter sweep scenario: the optimiser is a simple console application whose behaviour can be tuned by more than 25 parameters. The EMO application supports the optimisation of the well-known benchmark problems in the field [28] and can be extended by integrating the function to optimise for real life applications. In order to cover a reasonable domain of possible scenarios for real life cases, we generate 149 tasks by exploring different combinations of the following parameters:

Number of Generations: The number of iterations that the algorithm is repeated before terminating; the values range from 50 to 500 stepping by 50.

Number of Individuals: The number of sampling points used at each iteration to evaluate the function to optimise; the values: 50, 100, 200, 300, 500.

With this tuning, the requirements of the tasks in this BoT are *TFSize* (120 KB), *ETCPUTime* (0.002–30 min), and *OFSize* (0.47–57.71 KB). 78.52% of the tasks have *ETCPUTime* $\leq$ 5 min. The remaining tasks can be considered as average- or coarse-grain as their *ETCPUTime* ranges up to a maximum of 30 min.

In our meta-scheduler, the tasks in both the applications are categorised according to: $TFSize_{CI} = 1$ KB, $ETCPUTime_{CI} = 1$ min, and $OFSize_{CI} = 500$ KB.

5.3. Performance analysis

Three phases of experiments are conducted for our performance analysis.

5.3.1. Experiment phase I

The goal of this phase is to analyse and compare the overhead involved in individual task deployment with group-based task deployment. We select 50 tasks with *ETCPUTime* $\leq$ 1 min and *OFSize* $\leq$ 16 KB from the non-parametric sweep BoT for further deployment on R_0. It involves 10 experiments and the task granularity for each experiment is indicated in Table 3.

Table 3
Task granularities for experiment phase I.

Experiment	I	II	III	IV	V	VI	VII	VIII	IX	X
Task granularity	1	2	4	6	8	10	12	14	16	18
Total task groups	50	25	13	9	7	5	5	4	4	3
Total file transmissions	100	50	26	18	14	10	10	8	8	6

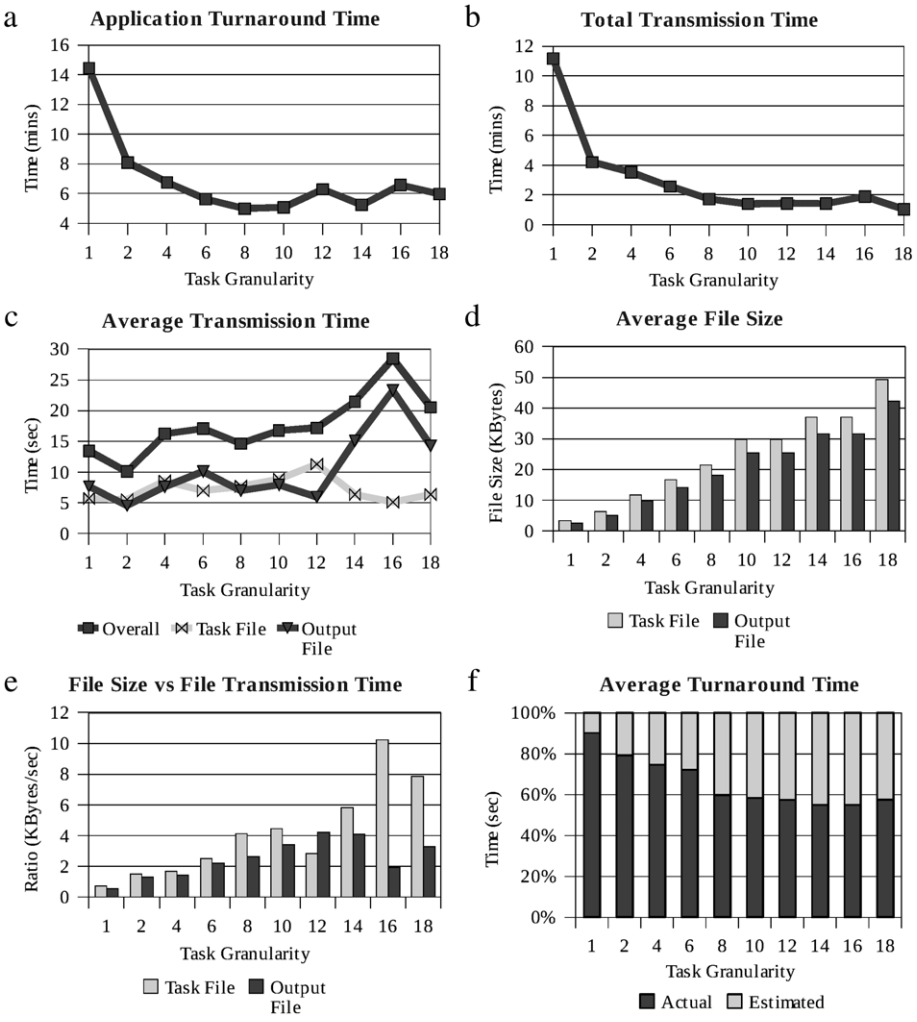


Fig. 4. Performance charts from experiment phase I.

Experiment I reflects the conventional task deployment where the tasks are transmitted to R_0 one-by-one; the task granularity is set to one. This induces 50 task executions and 100 file transmissions (50 for task file and 50 for output file transmissions).

In Experiment II, two tasks are grouped together before the deployment process. This incurs 25 task groups and 50 file transmissions. In Experiment X, 18 tasks are grouped into a batch. Thus, only three task groups are created and sent to R_0 for executions. Throughout these experiments, all the eight nodes of R_0 were free of processing loads, thus we could occupy the entire PBS queue of the cluster.

Fig. 4 shows the performance charts of the 10 experiments. Fig. 4(a) indicates that the conventional method consumes 14.45 min, whereas grouping two tasks before the deployment consumes 8.08 min. This shows a performance improvement of 44.09%. The minimum application turnaround time is achieved with an improvement of 65.38% when the granularity is eight (with seven task groups). One of the factors contributing towards this achievement is the utilisation of seven processing nodes in parallel, where one task group is assigned to one node.

Fig. 4(b) presents the transmission time involved in each experiment. The conventional method with 100 file transmissions spent 11.18 min, in contrast with a total of 1.71 min spent by the seven task groups of granularity eight. The transmission time is further analysed in terms of average time as shown in Fig. 4(c). We notice that, when the granularity is 16, the average output file transmission time reaches the peak with 23.32 s. However, there are only four output files to be transmitted, thus the overall transmission time still shows a better performance of 83.03% as compared to the conventional method.

Fig. 4(d) and (e) show the average task and output file size, and the resulting file size-transmission time ratio respectively. Despite the fluctuating network condition, the ratio reveals that an average of 0.73 KB of the task file are handled in one second during the conventional task deployment. On the other hand, the four task groups (granularity 16) are handled at 10.25 KB/s. This reveals that a better network utilisation can be achieved if we group the small files before the transmission and deployment activities.

The exact execution time of the 50 tasks in each experiment ranges from 8.69 min to 8.92 min. The 'actual' turnaround

Table 4
Resource-network utilisation constraints.

Utilisation constraints	Set I			Set II		
	R_1	R_2	R_3	R_1	R_2	R_3
MaxCPUTime (min)	2	5	4	10	10	8
MaxWCTime (min)	7	10	8	20	30	15
MaxSpace (MB)	8	10	15	10	15	20
MaxTransTime (min)	4	6	5	4	6	5
PCost (cost units per ms)	5	4	4	5	4	4

time of a task is (transmission time + execution time + other overheads), whereas the 'estimated' turnaround time is (execution time + transmission time). In order to realise the impact of task grouping on the overall application processing overhead, we computed the average 'actual' turnaround time and average 'estimated' turnaround time of all the tasks. Then, we scaled the summation of average 'actual' and 'estimated' turnaround time to 100% as shown in Fig. 4(f).

As for the conventional method, the 'estimated' time is only 10% of the 100%. The 'actual' time is 9 times more than the 'estimated' time. Better performance is observed starting from granularity eight with 'actual' time 60% and 'estimated' time 40%. The minimum overhead is achieved with granularities 14 and 16 with the 'actual' time being 55% and 'estimated' time being 45%.

5.3.2. Experiment phase II

Upon realising the need for task grouping, in this section, we test the performance of the proposed meta-scheduler based on resource-network utilisation constraints (Policies 1–4) and task availability (Policy 7). We plan four experiments for performance comparisons:

Experiment I: Conventional scheduling where a task is deployed as a resource or a cluster node becomes available.
Experiment II: Grouping is done based on the policies with the maximum task granularity set to 20.
Experiment III: Grouping is done based on the policies with no limit imposed on the task granularity.
Experiment IV: Grouping is done based on the policies for resources with extended utilisation constraints.

Experiment I is conducted using the conventional task deployment method. Experiments II–IV follow the proposed approaches and process flow shown in Section 4. All the 634 tasks from the non-parametric sweep BoT are used in this phase.

At the time of the experiments, the entire PBS queue of R_0 was reserved for other users and on average, only six nodes were available from R_1. Hence, in total we used eight processing nodes from R_1, R_2, and R_3 throughout the experiments. Table 4 shows the utilisation constraints imposed on the three resources. Set I constraints are used in Experiments II and III, while the extended constraints in Set II are used in Experiment IV.

During Experiments II–IV, the meta-scheduler arranges the 634 tasks into 25 categories based on the class intervals mentioned in Section 5.2. The resulting task categories are:

0–328, 1–59, 2–32, 3–16, 4–21, 5–12, 6–12, 7–7, 8–9, 9–10, 10–8, 11–9, 12–16, 13–12, 14–8, 15–6, 16–8, 17–6, 18–3, 19–14, 20–14, 21–9, 22–6, 23–6, 24–3.

For example, 0–328 indicates 328 tasks in category 0.

The first 30% dominating categories are 0, 1, 2, 4, 3, 12, 19. Two tasks from each dominating category are deployed on every resource during the benchmark phase; 14 tasks per resource. Hence, in total, 42 tasks are processed by the three resources before the first average analysis iteration.

Table 5 shows the number of groups and the corresponding task counts deployed on the resources during the experiments. In

Table 5
Task deployment, experiment phase II.

Resource	Deployment	Experiment			
		I	II	III	IV
R_1	Groups	301	193	166	89
	Tasks	301	388	352	426
R_2	Groups	109	35	32	25
	Tasks	109	58	83	63
R_3	Groups	224	49	47	31
	Tasks	224	188	199	145
Total	Groups	634	277	245	145
	Tasks	634	634	634	634

Experiment II, the meta-scheduler limits the maximum number of tasks in a group to 20 and in total, 277 task groups are formed at runtime. In Experiment III, no such condition is imposed, hence, 245 groups with higher granularities are deployed on the resources. When the resource *MaxCPUTime*, *MaxWCTime*, and *MaxSpace* are extended to support higher granularities in Experiment IV, in total, only 145 groups are created by the meta-scheduler.

The group count indicates the total task and output file transmission iterations to and from the resources. Experiment IV involves only 145×2 file transmission iterations as compared to 634×2 transmission iterations in the conventional deployment. The task count conveys the granularity of the groups (the number of tasks) processed by each resource. For example, in Experiment III, R_1 (with six active nodes) processed 166 groups with 352 tasks which is 55.52% of the total tasks; R_2 handled 32 groups with 83 tasks; and R_3 executed 47 groups with 199 tasks. Out of the participating eight processing nodes, R_3 delivers a better performance as it completed 31.39% of the tasks. This is due to the domain of R_3 which is same as the meta-scheduler and R_3 is a dedicated machine for our experiments.

The performance charts of Experiment Phase II are presented in Fig. 5. Conventional task deployment without any prior planning or analysis consumes 377.83 min to complete the 634 tasks. Experiment II–IV spent 51.20%, 48.47%, and 67.51% of the conventional time respectively as conveyed in Fig. 5(a). The minimum application turnaround time is achieved in Experiment III when the task granularity is decided merely based on the resource-network utilisation constraints. Table 6 lists the granularity for each resource during Experiment III and the number groups created at runtime. For example, for R_2, apart from the 14 benchmark tasks, only 18 task groups are created; 12 groups hold one task each (12 : 12); one group has two tasks (1 : 2); three groups contains nine tasks each (3 : 27); and two groups has 14 tasks each (2 : 28).

On the other hand, when the resource-network utilisation constraints are extended in Experiment IV, we notice that the turnaround time has increased to 255.08 min. Grouping many tasks until reaching the extended constraints (*MaxCPUTime*, *MaxWCTime*, and *MaxSpace*) results in course-grain groups. Each node will handle fewer groups with higher granularities. This reduces the optimal degree of parallelism suitable for this BoT. In addition, course-grain task groups increases the overhead at the resource site: task packing and unpacking time, and task waiting time. We observed that, on average, the wall-clock time of a task in Experiment IV is 13.61 min and the relevant overhead is 6.25 min. In contrast, as presented in Fig. 5(c), Experiment III reduces the overhead up to 89.44%.

Another interesting observation is the processing cost which depends on the task CPU time and the charges imposed by the resources. Fig. 5(d) shows the total task CPU time utilised at the resources (as for R_1, the average CPU time consumed at each node is computed). When handling the higher granularity tasks with

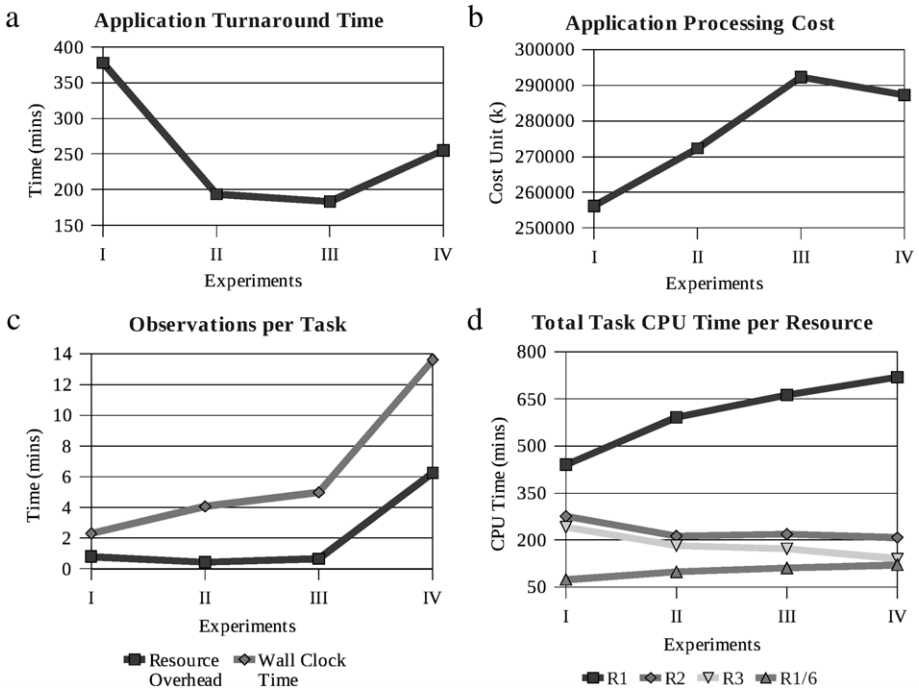


Fig. 5. Performance charts from experiment phase II.

Table 6
Task deployment, experiment III.

Granularity	Group:Task		
	R_1	R_2	R_3
1 (Benchmark)	14:14	14:14	14:14
1	116:116	12:12	12:12
2	8:16	1:2	–
3	–	–	7:21
4	10:40	–	1:4
5	6:30	–	4:20
6	1:6	–	3:18
7	3:21	–	2:14
9	–	3:27	1:9
12	1:12	–	–
13	4:52	–	–
14	–	2:28	–
15	3:45	–	–
29	–	–	3:87
Total Groups:Tasks	166:352	32:83	47:199

lower degree of parallelism, the nodes spend more time for task decompression, I/O operations, and output compression. Hence, the total CPU time increases in Experiments II–IV as compared to the conventional task deployment. The impact can be seen on the resulting processing cost in Fig. 5(b). In short, we can conclude that Experiment III provides the best application turnaround time when the grouping is done based on the resource-network utilisation constraints. However, QoS requirements need to be considered in order to utilise the resources in an economical manner.

5.3.3. Experiment phase III

In Experiment Phase II, we realise the need for considering QoS requirements when grouping the tasks. Hence, in this Experiment Phase III, the meta-scheduler considers all the seven policies pertaining to resource-network utilisation constraints, QoS requirements, and task availability. The policies and task grouping approaches are tested on the EMO application using three experiments:

Experiment I: Conventional scheduling where a task is deployed as a resource or a cluster node becomes available.

Experiment II: Grouping is done based on the policies with no limit imposed on the task granularity.

Experiment III: Grouping is done based on the policies for resources with extended utilisation constraints.

Resource-network utilisation constraints similar to those shown in Table 4 are used for this phase. Set I constraints are applied for Experiment II and Set II constraints for Experiment III.

During Experiments II and III, the meta-scheduler produced 19 categories comprising the 149 tasks. Two tasks from 20% of the dominating categories are selected for the benchmark phase. Eventually 64 groups are created in Experiment II and 56 groups in Experiment III.

Fig. 6 shows the resulting time and cost consumptions upon performing task grouping based on the seven policies. The conventional deployment required 96.81 min to complete all the 149 tasks in the BoT. Experiment II consumed 73.48 min, revealing a performance improvement of 24.10%. Experiment III with extended constraints used 89.77 min which is only 7.27% better than the conventional method.

Chart (b) shows the total processing cost charged for the three experiments. Experiment II results in the minimum cost which saves 11.01% of the amount spent for the conventional method. Meanwhile, Experiment III could save up to 2.9%.

It can be noticed that when grouping the tasks according to the extended utilisation constraints, we should limit the number of tasks in the group in order to achieve the minimum application processing time and cost. This is a major concern in a commercial grid where the users reserve the resources in advance [29]. The users have limited budget to process all the application tasks within the reserved period. In such a scenario, heuristic methods can be used to adaptively limit the task granularity by learning the grid status, and predicting the time and budget utilisation prior to the task grouping and deployment.

The design of the meta-scheduler can be extended to accommodate a service-oriented grid environment [30] in which grid services are available through web service interfaces (e.g. the Open Grid Service Architecture (OGSA)) [31,32]. The meta-scheduler can cooperate with the grid data management service (e.g. Globus Reliable File Transfer (RFT) [33]) for transmitting task and output files to and from the grid resources. In addition, the grid monitoring and

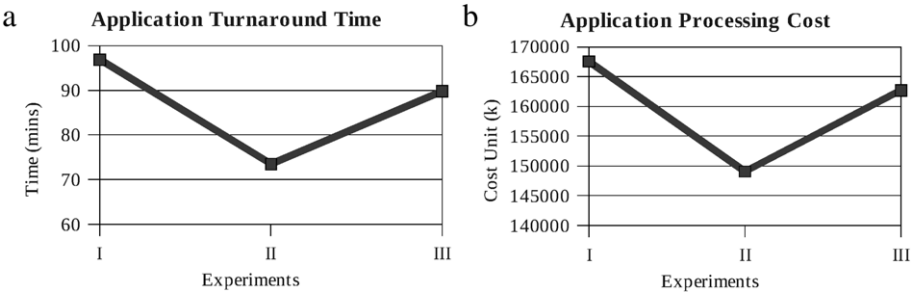


Fig. 6. Performance charts from experiment phase III.

information service, and the resource allocation manager service can be used for learning the specification of the grid resources, executing the tasks, and for monitoring the task progress [34]. During the deployment, multiple task can be grouped into a single task execution service [32] according to the task granularity policies proposed in this paper.

6. Conclusion

In this paper, we proposed policies and approaches for deciding the task granularity at runtime based on resource-network utilisation constraints, QoS requirements, and the latest average task deployment metrics. Our idea is implemented in a meta-scheduler which is tested in a real grid environment. The meta-scheduler deals with individual and clustered nodes without any synchronisation overhead. The approaches in the meta-scheduler support both parametric and non-parametric sweep BoTs. The experiments show that our proposed approaches lead towards an efficient and economical way of utilising the grid resources.

Our approach can be adapted to accommodate work-flow applications, data-intensive applications, and task migrations. In addition, the policies and task grouping can be coordinated with grid resource reservation techniques in order to achieve a better resource utilisation within the reserved or allocated time slots. The policy set given in this paper can be extended to control memory usage and total running or waiting tasks as desired by the resource providers.

Acknowledgements

This paper is an extended version of ICA3PP 2010 [16]. Here, we would like to acknowledge e-ScienceFund, Ministry of Science, Technology, and Innovation (MOSTI), Malaysia, and Endeavour Awards, Department of Innovation, Industry, Science and Research (DIISR), Australia, for supporting the research work and the development of the meta-scheduler described in this paper. We would also like to thank MIMOS for providing resources for our experiments.

Role of the funding source

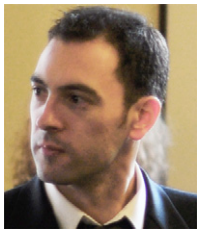
The fund from MOSTI was used to initiate this project in terms of resource set-up, consultations, simulations, and conference paper publications. The awards from DIISR gave the opportunity to conduct the extended research work (towards the development of the meta-scheduler) at the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Dept. of Computer Science and Software Engineering, The University of Melbourne, Australia.

References

[1] C. Kesselman, I. Foster, The Grid: Blueprint for a New Computing Infrastructure, 2. a. ed., Morgan Kaufmann Publishers, 2003.
[2] M. Baker, R. Buyya, D. Laforenza, Grids and grid technologies for wide-area distributed computing, Software: Practice and Experience 32 (15) (2002) 1437–1466.
[3] B. Jacob, M. Brown, K. Fukui, N. Trivedi, Introduction to Grid Computing, IBM Publication, 2005.
[4] F. Berman, G.C. Fox, A.J.G. Hey (Eds.), Grid Computing — Making the Global Infrastructure a Reality, Wiley and Sons, 2003.
[5] S. Venugopal, R. Buyya, W. Lyle, A grid service broker for scheduling e-science applications on global data grids, Concurrency and Computation: Practice and Experience 18 (2006) 685–699.
[6] R. Buyya, S. Date, Y. Mizuno-Matsumoto, S. Venugopal, D. Abramson, Neuroscience instrumentation and distributed analysis of brain activity data: a case for escience on global grids: Research articles, Concurrency and Computation: Practice and Experience 17 (15) (2005) 1783–1798.
[7] E.G. Coffman Jr., M. Yannakakis, M.J. Magazine, C. Santos, Batch sizing and job sequencing on a single machine, Annals of Operation Research 26 (1–4) (1990) 135–147.
[8] T. Cheng, M. Kovalyov, Single machine batch scheduling with sequential job processing, IIE Transactions 33 (5) (2001) 413–420.
[9] G. Mosheiov, D. Oron, A single machine batch scheduling problem with bounded batch size, European Journal of Operational Research 187 (3) (2008) 1069–1079.
[10] H. James, K. Hawick, P. Coddington, Scheduling independent tasks on meta-computing systems, in: Proceedings of Parallel and Distributed Computing Systems, Fort Lauderdale, US, 1999, pp. 156–162.
[11] A.C. Sodan, A. Kanavallil, B. Esbaugh, Group-based optimizaton for parallel job scheduling with scojo-pect-o, in: Proceedings of the 22nd International Symposium on High Performance Computing Systems and Applications, IEEE Computer Society, Washington, DC, USA, 2008, pp. 102–109.
[12] K.E. Maghraoui, T.J. Desell, B.K. Szymanski, C.A. Varela, The internet operating system: Middleware for adaptive distributed computing, International Journal of High Performance Computing Applications 20 (4) (2006) 467–480.
[13] W.K. Ng, T. Ang, T. Ling, C. Liew, Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing, Malaysian Journal of Computer Science 19 (2) (2006) 117–126.
[14] J.H. Stokes, Behind the benchmarks: Spec, gflops, mips et al. http://arstechnica.com/cpu/2q99/benchmarking-2.html, 2000.
[15] N. Muthuvelu, I. Chai, E. Chikkannan, An adaptive and parameterized job grouping algorithm for scheduling grid jobs, in: Proceedings of the 10th International Conference on Advanced Communication Technology, volume 2, 2008, pp. 975–980.
[16] N. Muthuvelu, I. Chai, E. Chikkannan, R. Buyya, On-line task granularity adaptation for dynamic grid applications, in: Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel Processing, volume 6081, 2010, pp. 266–277.
[17] N. Muthuvelu, I. Chai, E. Chikkannan, R. Buyya, Batch resizing policies and techniques for fine- grain grid tasks: the nuts and bolts, Journal of Information Processing Systems 7 (2) (2011) 299–320.
[18] J. Feng, G. Wasson, M. Humphrey, Resource usage policy expression and enforcement in grid computing, in: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, IEEE Computer Society, Washington, DC, USA, 2007, pp. 66–73.
[19] R.G.O. Arnon, Fallacies of distributed computing explained. http://www.webperformancematters.com/, 2007.
[20] N. Ranaldo, E. Zimeo, A framework for qos-based resource brokering in grid computing, in: Proceedings of the 5th IEEE European Conference on Web Services, the 2nd Workshop on Emerging Web Services Technology, Vol. 313, Halle, Germany, Birkhuser, Basel, 2007, pp. 159–170.
[21] M. Rahman, R. Ranjan, R. Buyya, Cooperative and decentralized workflow scheduling in global grids, Future Generation Computer Systems 26 (5) (2010) 753–768.
[22] B. Lowekamp, B. Tierney, L. Cottrell, R.H. Jones, T. Kielmann, M. Swany, A hierarchy of network performance characteristics for grid applications and services, June 2003.
[23] P. Huang, H. Peng, P. Lin, X. Li, Static strategy and dynamic adjustment: an effective method for grid task scheduling, Future Generation Computer Systems 25 (8) (2009) 884–892.
[24] Malaysian institute of microelectronic systems (mimos), http://www.mimos.my/.
[25] Enabling grids for e-science (egee), http://www.eu-egee.org/.
[26] M. Kirley, R. Stewart, Multiobjective evolutionary algorithms on complex networks, in: Proceedings of 4th International Conference Evolutionary Multi-Criterion Optimization, in: Lecture Notes Computer Science, vol. 4403, 2007.
[27] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1989.

[28] K. Deb, L. Thiele, M. Laumanns, E. Zitzler, Scalable test problems for evolutionary multi-objective optimization, in: Evolutonary Multiobjective Optmization, Springer-Verlag, 2005, pp. 105–145.
[29] E. Elmroth, J. Tordsson, Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions, Future Generation Computer System 24 (6) (2008) 585–593.
[30] I. Foster, C. Kesselman, J.M. Nick, S. Tuecke, Grid services for distributed system integration, Computer 35 (6) (2002) 37–46.
[31] V. Stankovski, M. Swain, V. Kravtsov, T. Niessen, D. Wegener, J. Kindermann, W. Dubitzky, Grid-enabling data mining applications with dataminninggrid: An architectural perspective, Future Generation Computer System 24 (4) (2008) 259–279.
[32] T. Glatard, J. Montagnat, D. Emsellem, D. Lingrand, A service-oriented architecture enabling dynamic service grouping for optimizing distributed workflow execution, Future Generation Computer Systems 24 (7) (2008) 720–730.
[33] R.K. Madduri, C.S. Hood, W.E. Allcock, Reliable file transfer in grid environments, in: Proceedings of the 27th Annual IEEE Conference on Local Computer Networks, IEEE Computer Society, Washington, DC, USA, 2002, pp. 737–738.
[34] G. von Laszewski, J. Gawor, C.J. Peña, I. Foster, Infogram: a grid service that supports both information queries and job execution, in: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society, Washington, DC, USA, 2002, pp. 333–342.

Nithiapidary Muthuvelu received her B.IT degree from Universiti Tenaga Nasional, Malaysia, in August 2003 and M.IT degree from the University of Melbourne, Australia, in December 2004. She is teaching at Multimedia University, Malaysia, since 2005. Currently, she is pursuing her Ph.D study in the field of grid computing at Multimedia University. Her research interests include: Distributed and Parallel Processing, Genetic Algorithms, and Data Communication. She is a member of the IEEE Computer Society.



Christian Vecchiola is a Postdoctoral fellow at Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computer Science and Software Engineering, the University of Melbourne, Australia. His primary research interests include: Grid/Cloud Computing, Distributed Evolutionary Computation, and Software Engineering. Since he joined the CLOUDS Laboratory, he focused his research activities and development efforts on two major topics: middleware support for Cloud/Grid Computing and distributed support for evolutionary algorithms. Dr Vecchiola completed his Ph.D. in 2007 at the University of Genova, Italy with a thesis on providing support for evolvable Software Systems by using Agent Oriented Software Engineering. He has been actively involved in the design and the development of the AgentService, which is a software framework for developing distributed systems based on Agent Technology.

Ian Chai received his B.Sci. and M.Sci. in Computer Science from the University of Kansas and his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. Since 1999, he has taught at Multimedia University in Cyberjaya, Malaysia.

Eswaran Chikkannan received his B.Tech, M.Tech, and Ph.D degrees from the Indian Institute of Technology Madras, India where he worked as a Professor in the Department of Electrical Engineering until January 2002. Currently he is working as a Professor in the Faculty of Information Technology, Multimedia University, Malaysia. Dr. C. Eswaran served as a visiting faculty and research fellow in many international universities. He has supervised successfully more than 25 Ph.D/M.S students and has published more than 150 research papers in reputed International Journals and Conferences. Prof. C. Eswaran is a senior member of IEEE.

Rajkumar Buyya is Professor of Computer Science and Software Engineering and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft Pty Ltd., a spin-off company of the University, commercialising its innovations in Grid and Cloud Computing. He has authored and published over 300 research papers and four text books. The books on emerging topics that Dr. Buyya edited include, High Performance Cluster Computing (Prentice Hall, USA, 1999), Content Delivery Networks (Springer, Germany, 2008) and Market-Oriented Grid and Utility Computing (Wiley, USA, 2009). He is one of the highly cited authors in computer science and software engineering worldwide (h-index=48, g-index=104, 12500+ citations).

Software technologies for Grid and Cloud computing developed under Dr. Buyya's leadership have gained rapid acceptance and are in use at several academic institutions and commercial enterprises in 40 countries around the world. Dr. Buyya has led the establishment and development of key community activities, including serving as foundation Chair of the IEEE Technical Committee on Scalable Computing and four IEEE conferences (CCGrid, Cluster, Grid, and e-Science). He has presented over 200 invited talks on his vision on IT Futures and advanced computing technologies at international conferences and institutions in Asia, Australia, Europe, North America, and South America. These contributions and international research leadership of Dr. Buyya are recognised through the award of "2009 IEEE Medal for Excellence in Scalable Computing" from the IEEE Computer Society, USA. For further information on Dr. Buyya, please visit his cyberhome: www.buyya.com.