Chapter V

Proposed Scheduling Model
5.1 Introduction

This chapter presents the scheduling strategies with the algorithm in the grid framework for e-governance applications. After the advance reservation the scheduling strategies can be applied to this architecture. The GridSim toolkit is used to develop a grid resource broker that supports the scheduling strategies and to quantify the broker's ability to dynamically select resources at runtime depending on their availability, capability and cost. The broker supports the scheduling algorithms with the two different strategies as cost and time.

5.2 Design of Scheduler

Scheduling is an allocation of tasks (computational & communicational) to resources within a time frame. This will optimize the allocation of tasks to resources to achieve performance. Grid resources are heterogeneous and of unpredictable availability. The scheduler takes decision in such environments to find out which applications will have access to the computational resources, the amount and localization of resources to be used by the applications.

---

5.2.1 Grid Resource Broker

This section briefly explains the design features required by a grid. It provides the users with a perfect computing environment.

Four main features of a grid are, [103]

- Multiple Administrative Domains and Autonomy
- Heterogeneity
- Scalability
- Dynamicity or Adaptability

The grid includes the following steps:

- The integration of individual software and hardware components into a combined network resource.

- The deployment of:
  
  a) Low-level middleware to provide a secure and transparent access to resources.
  
  b) User-level middleware to application development and the aggregation of distributed resources.

- The development and optimization of distributed applications to take advantage of the available resources and infrastructure.
The figure 5.2 shows a layered grid architecture of the connection approval component.

The key components of a grid are: [103] [104]

- **Grid Fabric**: It consists of all the globally distributed resources that are accessible from anywhere on the Internet. These resources could be computers running a variety of operating systems as well as resource management systems such as Load Sharing Facility, databases, and special scientific instruments.

- **Core Grid Middleware**: It offers core services such as remote process management, co-allocation of resources, storage access, information...
registration and discovery, security, and aspects of quality of service such as resource reservation.

• **User-Level Grid Middleware**: This includes application development environments, programming tools, and resource brokers for managing resources and scheduling application tasks for executing on global resources.

• **Grid Applications and Portals**: Grid applications are typically developed using grid-enabled languages and utilities like message-passing interface. Grid Resource Broker is designed to operate in an environment that comprises a set of sites. Each providing access to a set of servers.
5.2.2 Grid Broker Architecture

The GridSim toolkit is used to simulate a grid environment. The simulated grid environment contains multiple resources and user entities with different requirements. The users create an experiment that contains an application specification and quality of service requirements. During simulation each user entity has its own application and quality of service requirement. It creates its own instance of the broker entity for scheduling gridlets on resources.

The figure 5.3 shows the broker entity architecture and its interaction with other entities in the e-governance applications.

The key components of the broker are: Interface, resource discovery and scheduling, scheduling flow manager (packed with scheduling algorithm), dispatcher, and gridlets receptor.

1. The user entity contains the application description and sends user requirements to the broker through the interface.
2. The broker resource discovery and scheduling module interacts with the GridSim GIS entity to identify the contact information of the resources, and then interacts with resources. It creates a broker resource list which includes a placeholder for maintaining resource properties, a list of Gridlets committed for execution on the resource, and the resource performance data as predicted through the measurement and extrapolation methodology.
(3) The scheduling flow manager is used for mapping Gridlets to resources depending on the user’s requirements.

(4) The dispatcher selects the number of Gridlets for each of the resources. It can be staged for execution according to the usage guidelines to avoid overloading resources with single user jobs.

(5) The dispatcher then submits Gridlets to resources using the GridSim’s asynchronous service.

(6) When the Gridlet processing gets completed, the resource returns it to the broker’s Gridlet receptor module. Then it measures and updates the runtime
parameter and it serves in predicting the job utilization rate for making scheduling decisions.

(7) Steps 3–6 continue until all the Gridlets are processed. At the end, the broker returns updated experiment data along with processed Gridlets back to the user entity.

5.3 Implementation of Scheduler

The scheduler is designed by using the scheduling model. The scheduling algorithm is described within this model to improve the scheduling process.

5.3.1 Scheduling Model

A distributed e-governance computing environment, can be considered to consist of a set of M compute resources,

\[ R = \{r_1, r_2, \ldots, r_M\} \]

and

a set of P data hosts, \( D = \{d_1, d_2, \ldots, d_P\} \).

Here a compute resource is commonly a high performance computing platform such as processing nodes that are connected in a private local area network. A data host can be a dedicated storage resource such as a Mass Storage Facility connected to the Internet.

At the very least, it may be a storage device attached to a compute resource in which case it inherits the network properties of the latter. It is important to note
that even in the second case, the data host is considered as a separate entity from the compute resource.

The figure 5.4 shows a simplified distributed computing environment consisting of four compute resources and an equal number of data hosts connected by links of different bandwidths.

The physical networks between the resources consist of entities such as routers, switches, links and hubs. However, the model in this thesis abstracts the physical network to consider the logical network topology. As shown in figure 5.4 each compute resource is connected to every other data host by a distinct network link.

![Figure 5.4 A data-intensive environment](image_url)
This logical link is denoted by \( \text{Link}(r_m, d_p) \), \( r_m \in R \), \( d_p \in D \). The bandwidth of the logical link between two resources is the bottleneck bandwidth of the actual physical network between the resources and is given by \( BW(\text{Link}(r_m, d_p)) \). Data is organized in the form of datasets. A dataset can be an aggregated set of files, a set of records or even a part of a large file. Information about the datasets and their location is available through a catalog such as the Storage Resource Broker Metadata Catalog [6].

This application is composed of a set of \( N \) jobs, \( J = \{ j_1, j_2, \ldots, j_N \} \). Here \( N \gg M \), the number of compute resources. Also, a job is the smallest unit of computation, which is not possible to divide a job into smaller sub-units. It is also associated with a set of \( K' \) datasets, \( F = \{ f_1, f_2, \ldots, f_{K'} \} \), which are distributed on members of \( D \).

Specifically, for a dataset \( f_k \in F \), \( D_{f_k} \subseteq D \) is the set of data hosts on which \( f_k \) is replicated. Also, \( D_{f_1} \) and \( D_{f_2} \) need not be pair-wise disjoint for every \( f_1, f_2 \in F \). In other words, a data host can serve multiple datasets at a time.

A job \( j \in J \) processes a subset of \( F \) of size \( K \) denoted by \( F^j \). Each job requires one processor in a compute resource for executing the job and data host for accessing each of the \( K \) datasets required by the job.

The compute resource and the data hosts thus selected are collectively referred to as the resource set. This is associated with the job and is denoted by \( S^j = \{ R^j, D^j \} \) where \( R^j \leq R \) is a singleton representing the compute resource.
selected for executing the job. D^j is an L-sized set of data hosts chosen for accessing the datasets required by the job.

Therefore, R^j = \{r\}, r \in R and D^j = \bigcup_{f \in F^j} D_f. Multiple datasets can be retrieved from a single data host, \( L \leq K \), the number of datasets required for the job. The figure 5.5 shows an example of such a job \( j \) that requires resources shown in figure 5.4.

Consider a job \( j \) that has been submitted for execution to a compute resource \( r \). The time spent in waiting in the queue on the compute resource is denoted by \( T_w(j, r) \).

![Figure 5.5 Job model](image)

The expected execution time of the job is given by \( T_e(j, r) \). \( T_w \) increases with increasing load on the resource. Likewise, \( T_e \) is the time spent purely on computational operations and depends on the processing speed of the individual
nodes within the compute resource. For each dataset \( f \in F^j \), the time required to transfer \( f \) from \( d_f \) to \( r \) is given by

\[
T_t(f, d_f, r) = \text{Response time}(d_f) + \frac{\text{Size}(f)}{\text{BW}(\text{Link}(d_f, r))}
\]

Response time\((d_f)\) is the difference between the time when the request was made to \( d_f \) and the time when the first byte of the dataset \( f \) is received at \( r \). This is a measure of the latency of the response and is therefore, an increasing function of the load on the data host. The estimated completion time for the job, \( T_{ct}(j) \), is the wall clock time taken for the job from submission till eventual completion and is a function of these three times.

Figure 5.6 shows two examples of data-intensive jobs with times involved in various stages shown along a horizontal time-axis. In this figure, for convenience, the time for transferring \( f_1, f_2, \ldots, f_k \) is denoted by \( T_{t1}, T_{t2}, \ldots, T_{tk} \) respectively.

The impact of the transfer time of the datasets is dependent on the manner in which the dataset is processed by the job. For example, Figure 5.6 shows a common scenario in which grid applications request and receives the required datasets in parallel before starting computation. In this case,

\[
T_{ct}(j) = T_w(j, r) + \max_{f \in F^j} (T_t(f, d_f, r)) + T_e(j, r)
\]

The number of simultaneous transfers on a link determines the bandwidth available for each transfer and therefore, the \( T_t \). Then it shows a more generic data processing approach in which some of the datasets are transferred completely prior to execution and the rest are accessed as streams during the execution.
The red areas show the overlap of computation and communication. The thesis focuses on the application model that requires all the datasets to be transferred to the actual compute resource before execution. Also, the impact of data transfer time is the highest in this model. However, it is possible that lessons learnt from scheduling these types of applications may also be applicable to the other types of data-intensive applications.

In this system, there are time associated with the access, transfer and processing of data. The processing time is levied upon the computational service provider, while the transfer cost comes on account of the access cost for the data host and the cost of transferring datasets from the data host to the compute resource through the network.
The cost of executing the job \( j \) on the compute resource \( r \) is denoted by \( C_e(j, r) \) and the cost of transferring the dataset \( f \in F \) from \( d_f \in D_r \) to \( r \) by \( C_t(f, d_f, r) \) where \( C_t(f, d_f, r) = \text{Access cost}(d_f) + \text{Size}(f) \times \text{Cost}(\text{Link}(d_f, r)) \).

Here, \( \text{Access cost}(d_f) \) is the cost of requesting a dataset which is levied by the data host. It can be an increasing function on either the size of the requested dataset or the load on the data host or both. This cost regulates the size of the dataset being requested and the load which the data host can handle. The cost of transferring a unit size of the requested dataset through the network link between the data host and the compute resource is \( \text{Cost}(\text{Link}(d_f, r)) \). The cost of the link may increase with the Quality of Service (QoS) being provided by the network.

For example, in a network supporting different channels with different Quality of Services as described by Hui, et al. [119], the channel with a higher QoS may be more expensive but the data may be transferred faster. Hence, the file is transferred faster but at a higher expense. All traffic within a Local Area Network (LAN) is considered to be essentially free. Hence no cost is levied upon them.

Therefore, the total execution cost for job \( j \), \( C(j) \) is given by

\[
C(j) = C_e(j, r) + \sum_{f \in F} C_t(f, d_f, r)
\]
The notations that have been presented till now are summarised in table 5.1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>R={r_m}^M_{m=1}</td>
<td>Set of M compute resources</td>
</tr>
<tr>
<td>D={d_p}^P_{p=1}</td>
<td>Set of P data hosts</td>
</tr>
<tr>
<td>Link(r,d)</td>
<td>Logical link between r ∈ R and d ∈ D</td>
</tr>
<tr>
<td>BW(Link(r,d))</td>
<td>Available bandwidth of Link(r,d). It is the bottleneck bandwidth of the actual physical network between r and d</td>
</tr>
<tr>
<td>Cost(Link(r,d))</td>
<td>Price of moving a unit size of data (in MB or GB) through Link(r,d)</td>
</tr>
<tr>
<td>F={f_k}^K_{k=1}</td>
<td>Set of K datasets required by the application</td>
</tr>
<tr>
<td>D_f</td>
<td>Subset of D on which f is replicated</td>
</tr>
<tr>
<td>J={j_n}^N_{n=1}</td>
<td>Set of N jobs created for the application</td>
</tr>
<tr>
<td>F_j</td>
<td>Set of K datasets required by j ∈ J</td>
</tr>
<tr>
<td>R_j</td>
<td>Singleton set representing the computer resource executing j ∈ J</td>
</tr>
<tr>
<td>D_j</td>
<td>For a job j, the set of L data hosts from which the K datasets are retrieved, L ≤ K</td>
</tr>
<tr>
<td>S_j</td>
<td>Resource set associated with j ∈ J</td>
</tr>
<tr>
<td>T_w(j,r)</td>
<td>Expected waiting time for job j in the batch queue at r</td>
</tr>
<tr>
<td>T_t(f,d,f,r)</td>
<td>Expected time for transferring f ∈ F_j from d_f ∈ D_f to r</td>
</tr>
<tr>
<td>T_e(j,r)</td>
<td>Expected execution time for job j on resource r</td>
</tr>
<tr>
<td>T_c(j)</td>
<td>Expected completion time for job j</td>
</tr>
<tr>
<td>C_e(j,r)</td>
<td>Expected execution cost for job j on r</td>
</tr>
<tr>
<td>C_t(f,d,f, r)</td>
<td>Expected cost of transferring dataset f ∈ F_j from d_f ∈ D_f to r ∈ S_j</td>
</tr>
<tr>
<td>C(j)</td>
<td>Expected total cost of executing j</td>
</tr>
</tbody>
</table>

Table 5.1 Notations
5.3.2 Scheduling Algorithm

The scheduling model followed in the thesis is offline or batch mode scheduling of a set of independent tasks [79]. The common problem of creating a schedule for a set of jobs to run on distributed resources is called list scheduling and is considered to be NP-complete [118]. Many approximate heuristics have been devised for this problem and a short survey of these has been presented by Braun, et al. [118]. The figure 5.7 shows a general scheduling algorithm for batch mode scheduling of a set of jobs based on the skeleton presented by Casanova, et al. [45].

```
while there exists unsubmitted jobs do
    Update the resource performance data based on job scheduled in previous intervals
    Update network data between resources based on current conditions
    foreach unsubmitted job do
        Match the job to a resource set to satisfy the objective function at the job level
        Sort the jobs depending on the overall objective
    end
repeat
    Assign mapped jobs to each computer resource heuristically
until all jobs are submitted or no more jobs can be submitted
Wait until the next scheduling event
end
```

Figure 5.7 A generic scheduling algorithm

The resource broker is able to identify resources that meet minimum requirements of the application such as architecture, operating system, storage threshold and data access permissions. The scheduling is carried out at time
intervals called scheduling events [78]. These events can be determined to either run at regular intervals or in response to certain conditions.

### 5.3.3 Cost and Time based scheduling

Based on the model presented so far, the deadline for the application (denoted by $T_{\text{Deadline}}$) can be expressed in terms of job execution time as $\max_j C_j$ $T_{et}(j) \leq T_{\text{Deadline}}$.

Resources that are in demand due to their higher capabilities are expected to be more expensive than others. It follows the use of cheaper resources. This minimizes the cost while using more expensive (and more capable) resources. The resources can be compared on the basis of CPU share.

Depending on the user-provided scheduling preference, two objective functions are as follows,

- **Cost minimisation**: The objective is to produce a schedule that causes least expense while keeping the execution time within the deadline provided.

- **Time minimisation**: Here, the jobs are executed in the fastest time possible for the execution acting as the constraint. Here, the same heuristic is applied to achieve both of the objective functions.

The figure 5.8 lists the algorithm for cost minimisation scheduling of e-governance applications. The scheduling loop is invoked at regular polling intervals until all the jobs are completed or until the deadline is exceeded.
while \( J \neq \emptyset \) OR \( T_{\text{current}} < T_{\text{Deadline}} \) do  
// If jobs created is not null, or the expected time is lesser than the deadline

foreach \( r \sum R \) do  // For every resources
  
  Calculate performance data on the basis of resource performance in the previous polling interval
end

foreach \( d \sum D \) do  // For every data host
  
  Update the network information

  Let \( R_d \leftarrow \{ r_m | r_m < r_{m+1} \text{ if } \text{Cost}(\text{Link}(d, r_m)) < \text{Cost}(\text{Link}(d, r_{m+1})) \} \)
end

// Mapping Begins

foreach \( j \sum J \) do  // For every jobs created for application
  
  Let \( S^i_j \leftarrow \{ R^i_j, D^i_j \} \), \( R^i_j \leftarrow \emptyset \), \( D^i_j \leftarrow \emptyset \)

  Let \( R^i_j_{\text{temp}} \leftarrow \emptyset \) // A temporary variable

  foreach \( f \sum F^j \) do  // For every dataset required by \( j \)
    
    Let \( U \leftarrow \{(d_f, r) | d_f \in D_f \text{ where } r \text{ is the first element of } R_{df} \}

    Find \( (d_f, r) \) such that \( C(t, d_f, r) + C_u(j, r) \) is minimum over \( U \)

    if \( S^i_j = \{ \emptyset, \emptyset \} \) then
      
      \( R^i_j \leftarrow \{ r \}, D^i_j \leftarrow \{ d_f \}, R^i_j_{\text{temp}} \leftarrow \{ r \} \)
    
    else
      
      \( R^i_j \leftarrow \{ r \} \cup \{ d_f \} \)
    
    end

  end

  \( S^i_j \leftarrow \min\{ \{ R^i_j \}, \{ R^i_j_{\text{temp}}, D_j \} \} \), \( R^i_j_{\text{temp}} \leftarrow R_j \)
end

end

// Mapping Ends

Dispatch(J, T_{\text{Deadline}})  // Dispatch with time deadline

Wait until next polling interval

Update resource by taking into account jobs completed in the last interval

end

Figure 5.8 An Algorithm for minimizing cost of scheduling

At every polling interval, the performance data of the compute resources is updated. It can be done by taking into account status of the jobs allocated to and those completed jobs by the resource in the previous intervals and information
from external performance monitors. This is used to calculate the limit of allocation (number of available job slots) of the resource for the current polling interval and also queried are market information services for latest information on instantaneous resource prices.

For each data resource, the cost and available bandwidth between itself and the computational resources is refreshed by querying the network information services. Then, for each data host, a sorted list of available compute resources is created based on the cost of transmitting a unit of data between the data host and the compute resource. This is followed by the mapping loop wherein each job is mapped to a set of resources. After the jobs are mapped, the dispatch function is invoked and the jobs are submitted to the selected resources while taking into consideration deadline constraints specified by the user.

The aim of the mapping loop is to match each job to a resource set and then assign the jobs to the selected resources. For each job, the loop starts off with an empty resource set \( S_j \) which itself is a set of the empty singleton \( R_j \) and the empty set of data hosts \( D_j \). For each dataset associated with the job, another set \( U \) is created. It consists of ordered pairs. Each of which has one data host that contains the dataset and a compute resource such that the cost of transfer for that dataset is minimum. The compute resource is the first element of the sorted set of compute resources \( (R_d) \) that has been created for each data host.

The ordered pair \( (d_r, r) \) that provides the smallest cost is then selected out of all the pairs in \( U \). The compute resource from the ordered pair is then assigned
to $R_j$ while the corresponding data host is added to $D_j$. $R_{j\text{temp}}$ is another singleton which has the compute resource selected in the previous iteration of the loop. A comparison is then made between the resource set with the current compute resource ($\{R_j, D_j\}$) and the previous compute resource ($\{R_{j\text{temp}}, D_j\}$). Finally the one which provides the least cost is then selected as the resource set for the next iteration of the dataset loop.

The matching heuristic is therefore, essentially a greedy strategy with a choice step to improve the resource set being selected in every iteration. This is a straightforward greedy choice for a job that requires a single dataset.

The figure 5.8 describes the process involved with multiple datasets. For each dataset, a pair of compute resource and data host is selected such that it ensures the best metrics for the job if only that dataset are involved.

This is merged with the resource set that has been built up previously to derive two resource sets, one with the compute resource selected in the previous iteration and the other with the current compute resource.

The resource with the exact limit is selected as the input for the next iteration. The idea behind this heuristic is to ensure the adding every pair of a compute resource and a data host produces a better resource set at the end of each iteration than that was produced by the previous iteration.
For each dataset $f_i$ do:

Best Compute/Data combination for current dataset giving least cost/time

Among the two, the resource set that delivers the least cost/time is selected and the process repeated with that set

Figure 5.9  The greedy matching heuristic

The figure 5.9 shows the job dispatch function. The allocated jobs are sorted in the ascending order of their expected costs for their respective resource sets. Then, starting with the job with the least cost, each job is submitted to its compute resource selected in the mapping step if the allocation for that resource has not been exhausted by previous assignments.

For cost minimisation, it is determined whether the deadline is violated by checking whether the current time ($T_{current}$) plus the expected completion time exceeds $T_{Deadline}$. If so, the job goes back into the unsubmitted list in the expectation that the next iteration of the mapping loop will produce a better resource set for that job. If expected time is exceeded by the current job, then the dispatching is halted and the functions returns to the main loop as the rest of the
jobs is with the greater waiting time and lesser cost is selected. It also checks if the resource needed is lesser than the available resources. If these two constraints are not violated, then the job is submitted to the compute resource and removed from the list of unsubmitted jobs.

```
Dispatch (J, TDeadline) // Dispatch the job with job and its deadline
Sort J in the ascending order of C(j), ∀j ∈ j
Expected Cost ← Cost spent // Estimate the cost
foreach j ∈ J do // For every job
    Take the next job j ∈ J in sorted order
    if r ∈ R can be allocated more jobs then
        if (Tcurrent + Tct(j) < TDeadline then
            submit j to r
        else stop dispatching and exit to main loop
    end
    Expected Cost=Expected Cost + c // Calculate the expected cost
    Remove j from J
end
```

**Figure 5.10  Deadline constrained job dispatch**

Figure 5.11 shows the time minimisation that can be achieved with the same algorithm but with time-specific variables. The mapping function sorts the compute resources for each data host based on the time for transferring unit data. That is, the termCost(\(\text{Link}(r,d)\)) in figure 5.8 is replaced by \(1/\text{BW}(\text{Link}(r, d))\).
while $J \neq \emptyset$ OR $T_{\text{current}} < T_{\text{Deadline}}$ do
// If jobs created is not null, or the expected time is lesser than the deadline
foreach $r \in R$ do // For every resources
  Calculate performance data on the basis of resource performance in the previous polling interval
end
foreach $d \in D$ do // For every data host
  Update the network information
  Let $R_d \leftarrow \{r_m | r_m < r_{m+1} \text{ if } \frac{1}{\text{BW}}(\text{Link}(d, r_m)) < \frac{1}{\text{BW}}(\text{Link}(d, r_m)) \forall r_m \in R, 1 \leq m \leq M \}$
end
// Mapping Begins
foreach $j \in J$ do // For every jobs created for application
  Let $S_j \leftarrow \{R_j, D_j \}$, $R_j \leftarrow \emptyset$, $D_j \leftarrow \emptyset$
  Let $R_j^{\text{temp}} \leftarrow \emptyset$ // A temporary variable
  foreach $f \in F$ do // For every dataset required by $j$
    Let $U \leftarrow \{(d_f, r) | d_f \in D_f \}$ where $r$ is the first element of ordered set $R_d$
    Find $(d_f, r)$ such that $T_t(f, d_f, r) + T_e(j, r)$ is minimum over $U$
    if $S_j = \{\emptyset, \emptyset\}$ then
      $R_j \leftarrow \{r\}, D_j \leftarrow \{d_f\}, R_j^{\text{temp}} \leftarrow \{r\}$
    end
    else
      $R_j \leftarrow \{r\} \cup \{d_f\}$
    end
  end
  $S_j \leftarrow \min\{\{R_j, D_j\}, \{R_j^{\text{temp}}, D_j\}\}$, $R_j^{\text{temp}} \leftarrow R_j$
end
// Mapping Ends
Dispatch($J, T_{\text{Deadline}}$) // Dispatch with time deadline
Wait until next polling interval
Update resource by taking into account jobs completed in the last interval
end

Figure 5.11 An Algorithm for minimizing execution time

If the deadline is violated by the current job, then the dispatch function returns to the main loop. The cost and time based scheduling algorithm presented

115
here is implemented in the Gridbus broker. Details of the resources including configuration is provided in the next chapter.

5.4 Summary

Data grid environment consists of heterogeneous computational, storage and networking resources that are shared among the users and may have expenses associated with their usage. A scheduler operating in such an environment must not only take into account the variations of availabilities, capabilities and costs among the resources but also should consider application requirements that may include multiple large-sized datasets, each replicated on multiple resources. This chapter models this problem properly and applies it to cost and time based scheduling of distributed applications.