Cost Minimization Heuristics for Scheduling Workflows on Heterogeneous Distributed Environments

Presented by

María Alejandra Rodríguez Sossa

Submitted in partial fulfillment of the requirements of the course

Masters of Engineering in Distributed Computing

The Cloud Computing and Distributed Systems (CLOUDS) Laboratory

Department of Computer Science and Software Engineering

The University of Melbourne

November 2011

Supervised by Dr. Rajkumar Buyya

Acknowledgments

I would like to thank my supervisor Professor Rajkumar Buyya for providing me with the opportunity to work in this project and get hands on experience with technologies which are the result of years of research and development. He supervised the progress of the project and provided me with the feedback, resources and tools necessary to successfully fulfill the project's objectives.

I would also like to thank Suraj Pandey who provided valuable guidance throughout the project. He has a vast experience with the Cloudbus Workflow Management System and scheduling algorithms helped me gain a better understanding on the topics and accomplish the established goals.

Finally, I would like to thank the researchers at the CLOUDS lab that helped me by either providing their input or on understanding the architecture of Cloudbus Workflow Management System.

Abstract

Many large scale scientific problems require computing power that goes beyond the capabilities of a single machine. The data and compute requirements of these problems demand a high performance computing environment such as a cluster, a grid or a cloud platform in order to be solved in a reasonable amount of time. In order to efficiently execute workflows and utilize the distributed resources in an appropriate way, a scheduling policy needs to be in place.

Application schedulers are driven by different objectives such as minimization of the total execution cost, minimization of the total execution time, even utilization of the available resources (load balancing), a combination of these and so forth. This project focuses on minimizing the overall execution cost of workflow applications in heterogeneous high performance distributed environments.

Specifically, this work focuses on two different types of environments. The first one is a grid environment and assumes that a static set of resources is available for executing the application workflow. The second one is a cloud environment in which the resources are dynamically leased as they are needed by the application. Two algorithms, one for each scenario, are proposed and both are based on a Particle Swarm Optimization metaheuristic algorithm.

The algorithm developed for grid environments was integrated into the Cloudbus Workflow Management System and its performance was evaluated using different workflows of varying sizes. The approach was compared against a baseline round robin algorithm. The findings show that the proposed algorithm achieves an average 28.5% cost reduction while distributing the workflow tasks more evenly across the available resources.

Table of Contents

Table of Figures

1. Introduction

Many large scale scientific problems require computing power that goes beyond the capabilities of a single machine. The data and compute requirements of these problems demand a high performance computing environment such as a cluster, a grid or a cloud platform in order to be solved in a reasonable amount of time. Furthermore, workflows, which can be defined as a set of tasks and a set of dependencies between them, are a common way of expressing these scientific applications. In order to efficiently execute workflows and utilize the distributed resources in an appropriate way, a scheduling policy needs to be in place.

Application schedulers are driven by different objectives. Some of these objectives include the minimization of the total execution cost, minimization of the total execution time, even utilization of the available resources (load balancing), a combination of these and so forth. This project focuses on minimizing the overall execution cost of workflow applications in heterogeneous high performance distributed environments; goal that is achieved by using a Particle Swarm Optimization metaheuristic algorithm.

Particle Swarm Optimization (PSO) is a flexible technique widely used nowadays to solve optimization problems [1]. This algorithm was proposed by Kennedy and Ebehart in [2] and is based on the social behavior of bird and fish flocks; it simulates the behavior of individuals in the flock in order to maximize the survival of the species. PSO is similar to other population based algorithms such as Genetic Algorithms (GA) in that the system is initialized with a set of random solutions; however, it is different in that it relies on the social behavior of the particles instead of recombining the individuals of the population [3].

In broad terms, the PSO is based on a swarm of particles moving through space and communicating with each other in order to determine an optimal search direction. Furthermore, the movement of particles in the algorithm is a stochastic process guided by a given current velocity, the particle's own experience (local best) and that of the whole group (global best). PSO has better computational performance than other evolutionary algorithms [2] and fewer parameters to tune, which also makes it easier to implement. Many problems in different areas have been successfully addressed by adapting PSO to specific domains; for instance this technique was used in the reactive voltage control problem [4], pattern recognition [5] and data mining [6] among others.

This project investigates the workflow scheduling problem in two different scenarios. The first one is a grid environment in which the available computing resources and their characteristics are known in advance. The second one is a cloud environment in which there is no predefined set of resources and instead these need to be leased at run time in such way that the scheduling objective is met. Both of the proposed algorithms are based on PSO and their main objective is to minimize the execution cost of the workflow on the distributed resources.

1.1. Background

Applications with ever growing requirements have triggered the development of distributed systems; distributed computing has evolved from high performance computing to grid computing and now to the most recent paradigm, cloud computing. All these technologies have a goal in common, provide large scale applications with a powerful platform on which they can be deployed and executed as they might require for instance, computing power that goes beyond the capabilities of a single machine. Furthermore, they aim to provide a large number of users with a reliable, scalable and efficient service.

Over the years, the way in which this service is provided has evolved towards a utility model in which users are charged based on their consumption. One of the main scientists in charge of the ARPANET (Advanced Research Projects Agency Network) project foresaw this when he stated [28] "As of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of, 'computer utilities', which, like present electric and telephone utilities, will service individual homes and offices across the country".

1.1.1 Grid and Cloud Computing Platforms

Grid Computing

Based on a service oriented architecture, grid computing allows heterogeneous resources to be accessed in a secure and uniform way and enables the creation and management of virtual organizations (VOs) [29]. A virtual organization can be defined as individuals, institutions or organizations that share resources such as processing time, software and data in a controlled, secure and flexible way in order to achieve a common goal [30]. Therefore, through grid computing, distributed networked resources are integrated and coordinated allowing them to function as a single virtual whole. One of the main goals of this type of computing is to aggregate the processing power of the interconnected machines in order to solve large scale problems that cannot be handled by a single machine. Even though the processor is one of the most obvious resources shared in a grid, other such as storage systems, sensors and applications can also be shared.

There are various definitions of grid computing. For instance, the Globus project (Argonne National Laboratory, USA) [31] defines grid as "an infrastructure that enables the integrated, collaborative use of high-end computers, networks, databases, and scientific instruments owned and managed by multiple organizations." Another definition is that given by Dr. Buyya et al. is "Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed 'autonomous' resources dynamically at runtime depending on their availability, capability, performance, cost, and users' Quality of Service (QoS) requirements." [29].

A high level view of a grid environment is depicted in figure 1. Grid information services are used to register grid resources. The grid resource broker receives application requirements from end users and based on these, queries the grid information services in order to discover suitable resources. After this, the resource broke schedules and monitors the application on the selected resources until it completes its execution. Furthermore, grid environments offer additional services such as security, directory, resource allocation, execution management and resource aggregation among others [29].

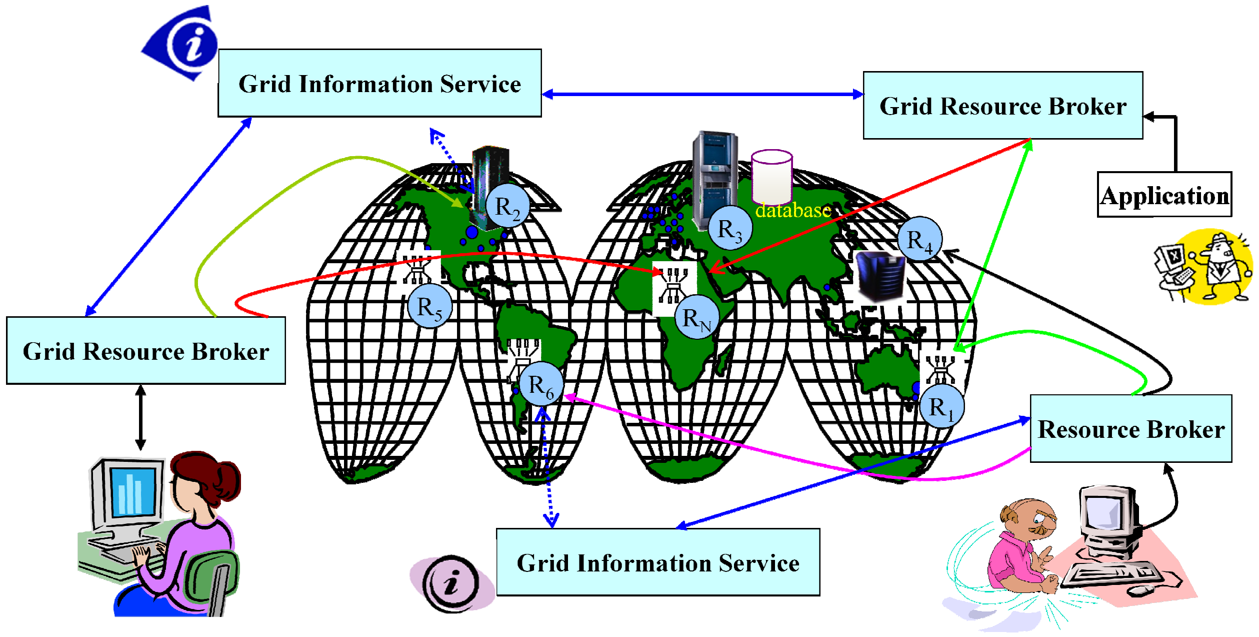


Figure 1 High level view of a global grid [29]

There are several challenges that need to be addressed by grid middleware systems in order to hide the complexity of the underlying distributed environment from the end users [32]. Firstly, a grid environment is heterogeneous by nature as resources may have a wide range of hardware and software. Secondly, grid resources might be spread across political and geographical boundaries and they might be under the control of different administration with different policies. Finally, the grid environment is dynamic in nature. This means that the availability and performance of the resources is unpredictable; for instance, requests from within an organization may have higher priority or different cost than requests from outside the organization.

These challenges have been addressed in various ways by different architectural approaches. One of such approaches is based on the creation of virtual organizations [33]; these VOs represent different real world organizations that ally in order to share resources and achieve a common goal. Each VO defines the assets that will be shared and a set of policies to access them; these policies specify how participants are allocated resources and they reflect the objectives of the VO.

Another architectural approach is based on economic principles [34], in this case, the owners or resource providers compete to provide the best service to the users or resource consumers. Users select the resources based on different criteria such as ability to meet specific requirements, price of the resources and quality of service (QoS) expectations. For instance a user may need to execute an application under a certain budget or before a specified deadline; both of these constraints are QoS requirements that drive the user when selecting an appropriate resource provider.

Grid computing however poses several limitations that bound users and applications deployed on them. In general, grid platforms are limited to a specific set of users and IT resources; furthermore, these resources are shared mostly on a volunteer manner with the sole purpose of achieving a common goal [35]. Cloud computing is a new flexible trend that overcomes all these limitations by delivering infrastructure and software as services in a pay per use basis.

Cloud Computing

Cloud computing is an emerging technology that enables the delivery of services over the Internet; specifically, it aims to deliver IT resources on a pay per use basis. In the modern society, utility services such as water, gas and electricity are massively deployed and they can be accessed from almost anywhere and at anytime; these services reach billions of people who are charged based on their consumption of the services. Cloud computing is part of the infrastructure that makes possible the use of computing resources under the utility model [37].

Through Cloud Computing, vendors make CPU cycles, storage and application hosting accessible under a service level agreement; these resources are exposed as standards based Web services and follow the utility pricing model as mentioned before [36]. The offered computing infrastructure is viewed as a “cloud” from which applications can be accessed from anywhere and on demand [38].

There are many definitions of Cloud Computing and its characteristics; Buyya et al. [38] describe it as “a parallel and distributed computing system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements (SLA) established through negotiation between the service provider and consumers”.

The main idea behind Cloud Computing is to provide computing, storage and software as-a-service [36]. Therefore, product offerings are classified into a hierarchy of as-a-service terms: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). Each of these classes represent a different abstraction level and as a whole, they can be understood as a layered architecture where the layers above leverage from the services provided by the layers below [36]; this is depicted in figure 2.

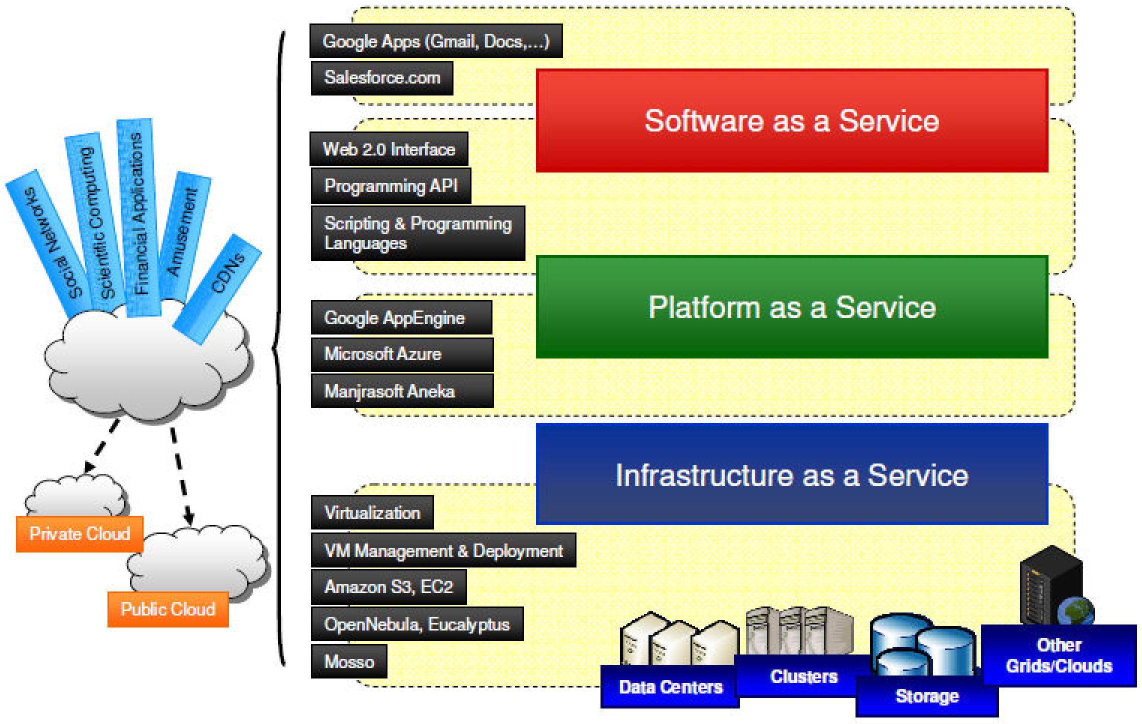


Figure 2 Classification of Cloud Computing Services [37]

At the bottom layer of the cloud computing stack is Infrastructure as a Service, IaaS. At this level, virtualized resources such as computation, storage and communication are provided. It is common to lease these resources by means of virtual machines so that users can deploy or run any software and in general, manage them in the same way they manage a physical machine. Amazon Web Service is an example of an IaaS provider.

The next layer in the stack is Platform as a Service, PaaS. Vendors who offer this type of service provide an environment on which developers can easily create and deploy applications. These applications can scale automatically as needed and therefore developers do not have to be aware of the amount of resources it will consume. Another features offered at this level are the provision of different programming models and specialized services that make the creation of applications simpler and more robust. Google App Engine is an example of PaaS.

Finally, at the top of the stack is Software as a Service, SaaS. It provides services or applications that can be accessed through the Web; this allows users to use on-line software instead of locally installed software. Salesforce.com [39], a SaaS provider that offers CRM applications, describes software as a service and its benefits as "a way of delivering applications over the Internet—as a service. Instead of installing and maintaining software, you simply access it via the Internet, freeing yourself from complex software and hardware management."

A Cloud Computing platform should be able to fulfill a set of basic features expected from its consumers, namely, a cloud should offer self service, per usage metered and billing, elasticity and customization [36]. Self service means that customers should be able to request, customize, pay and use services without intervention of human operators [40]. Per usage metered and billing implies that customers should be able to use and get charged only the necessary amount. Elasticity refers to the ability of increasing the number of resources at any time if the application requires it. Finally, in the case of IaaS, customizable means that the users should be able to deploy specialized applications and have privileged access to the virtual servers; in the case of PaaS and SaaS the level of customization is not so high as they offer less flexibility and control over the resources.

Despite the great acceptance that Cloud Computing has had in the recent years, a number of challenges and risks are inherent to it and should be taken into account by end users and providers. First, Cloud Computing is based on the use of third party services (from the user's point of view) which are used to host or manipulate sensitive data and to perform critical operations. Therefore, privacy and security are important challenges that need to be addressed by providers so that customers can trust them and their products. Second, because Cloud Computing is a recent technology, there are no established standards that dictate the way data or applications should be handled in cloud platforms; as a consequence, different platforms have different implementations that do not interoperate which leads to data and application lock in. Third, users have certain expectations when they move their resources to the cloud, in particular they expect availability, fault tolerance and disaster recovery; thus, a SLA that acts as a warranty must be arranged between clients and providers. Finally, an important challenge is the efficient management of virtualized resources; physical resources need to be shared efficiently among virtual machines which have different workloads [36].

To summarize, cloud and grid computing are distributed computing paradigms which enable the aggregation of valuable resources as well as the deployment and execution of large scale applications with requirements that cannot be met by a single machine. Cloud computing is considered to be the evolution of grid computing; unlike grid computing, clouds are meant for the masses and offer unlimited resources in an elastic way, charging on a pay per use basis. Table 1 summarizes the differences between both approaches.

Grid Computing	Cloud Computing
• Based on ability to negotiate resource-sharing arrangements • Coordinates independent resources • Uses open standards and interfaces • Allows for heterogeneity of computers • Resources distributed across large geographical boundaries	• Resources often leased from Cloud service providers on a pay-as-you-go (PAYG) basis • Resources are managed in virtualized data centers • Service provider uses virtualized resources that may be using services from multiple data centers

• Loose coupling of computers and storage services • Physical resource-sharing among large number of users • usually limits dynamic provisioning and scalability within a single administrative domain	• Service oriented • Dynamically provisioned, elastic and highly scalable

Table 1 Main differences between cloud and grid computing [35]

1.1.2 Workflow Scheduling

In broad terms, a workflow is a set of tasks with dependencies between them. The process of scheduling a workflow in a set of resources consists of mapping tasks or group of tasks to an available compute resource and scheduling their execution so that the dependencies between them are preserved. Most workflows can be expressed as a Directed Acyclic Graph (DAG); by definition, such graphs have no cycles or conditional dependencies. Figure 3 depicts a sample workflow represented as a graph.

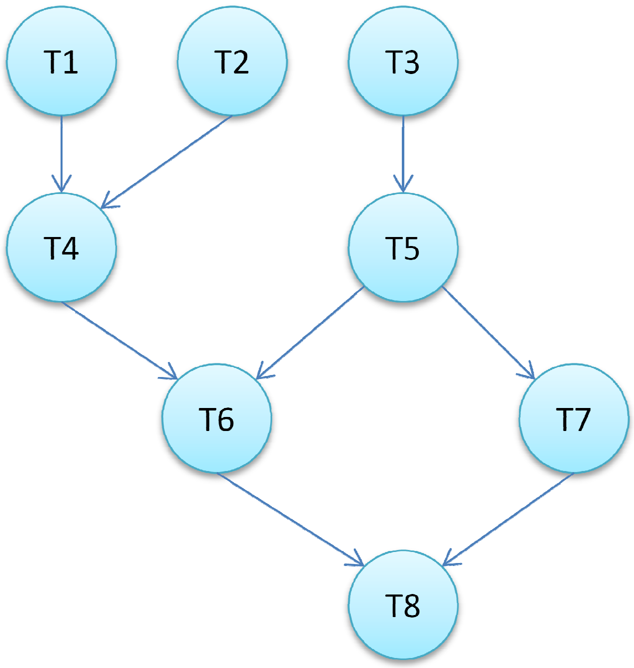


Figure 3 Sample workflow

Different architectures may support the workflow scheduling infrastructure. In particular, there are three major categories in which these architectures may fall: centralized, hierarchical and decentralized [41]. In the centralized scheme there is a unique, central scheduler that decides the mapping for every task in the workflow. The hierarchical approach on the other hand, defines a main scheduler which manages a set of low level schedulers in charge of a subset of the workflow tasks. Finally, in the decentralized architecture only a set of independent, decentralized schedulers exist and each of them is responsible of a sub workflow [42].

The decision to map a task to a resource can be made based on the information available to the scheduler. There are two main types of decisions, local and global. Local decisions are made based solely on the information of the single task (or group of tasks) being handled by a particular scheduler. Conversely, global decisions are those made considering the entire workflow as opposed to a single, isolated task.

The problem of finding an optimal workflow schedule in distributed environments has been widely studied and in most cases is an NP-complete problem [7]. In general, the scheduling process is driven by a QoS constraint imposed by the user; many heuristics have been proposed in order to obtain solutions that are near optimal and meet these QoS requirements. The QoS or objectives of the scheduling process vary from application to application; they include restrictions such as minimization of the overall execution cost, minimization of the total execution time or a combination of both among others.

1.2 Related Work

As mentioned earlier, a workflow can be represented as a DAG in which the nodes correspond to tasks and the edges to the dependencies between the tasks. A workflow scheduling process would then be responsible for managing the execution of these interdependent tasks on the heterogeneous distributed resources; particularly, it would be responsible of allocating each workflow task to a suitable resource so that the desired objective function is satisfied when the execution is completed. The job to resource allocation problem has been widely studied for many years and is known to be an NP-complete problem [7]. Such problems are those for which no known algorithm is capable of generating an optimal solution in polynomial time. The brute force approach would be to generate all possible schedules by trying all the possible task -resource combinations and select the optimal solution; however, the high overhead this technique implies makes of it an impractical approach. For this reason, more efficient heuristic based approximation algorithms have been used to address this problem.

The Greedy Randomized Adaptive Search Procedure (GRASP) is a metaheuristic for solving combinatorial optimization problems; in broad terms, an iterative randomized search is performed in order to find a good approximation to the optimal solution for a given problem [8]. This technique has been used to solve the job shop scheduling problem by [9] and to schedule workflows on grids by [10]. Furthermore, a scheduling algorithm for workflow allocation based on GRASP [11] is implemented in Pegasus [12], a framework for mapping complex workflows into grid resources proposed by Deelman et al.

Simulated Annealing (SA) [13] is another metaheuristic global optimum search technique. As its name implies, the technique is based on the steel and ceramic annealing process in which the materials are slowly heated and cooled in order to alter their physical properties. Lenstra et al. [15] investigates an approach to the job shop scheduling problem using an SA approach and finally, the authors in [14] implemented a simulated annealing scheduler based on their ICENI grid middleware and found that it outperformed other algorithms such as random and best of n random.

Genetic Algorithms (GA) [16] have also been used to solve the mentioned problem. Based on the principle of evolution, a genetic algorithm is basically a stochastic search technique that allows a near optimal solution to be derived from a large search space in polynomial time. This algorithm has been applied extensively to the resource allocation problem. For instance, [17], [18] and [19] use GAs to schedule workflows on homogeneous distributed environments whereas [20] uses them for scheduling workflows on heterogeneous resources. Additionally, ASKALON [21], a grid environment for the execution of scientific workflow applications uses genetic algorithms for scheduling purposes [22].

In addition to this work, there have been some comparative studies that demonstrate that PSO performs better than GAs in when applied to the mentioned scheduling problem. The authors of [23] experimented with PSO and a GA for solving the task assignment problem in an homogeneous environment and concluded that PSO converges faster than the implemented GA. Additionally, the research in [24] concludes that a PSO based approach is capable of generation better schedules than GA based one. Even though both approaches are valid and similar to each other, PSO was chosen in this project for several reasons. Firstly, studies demonstrate that it is faster than GA. Secondly, it has fewer operators to define, in GA for instance the reproduction, crossover and mutation functions need to be defined and hence makes of application's performance more dependant to the fine tuning of these parameters. Finally, the nature of the problem makes it simple to use discrete numbers in PSO so that the particle's position can easily be associated to task-resource mappings.

1.3 Aim

This project aims to develop two algorithms:

1. Propose a scheduling heuristic that finds a task to resource mapping in such way that the total execution cost of the workflow application is minimized and the available resources are evenly utilized. In other words, the algorithm will find a balance between the minimum execution cost and an even load distribution among all the resources.

2. Develop a set of heuristics that consider the elastic cloud leasing model in order to decrease the cost of executing workflows on an IaaS cloud provider. The algorithm will dynamically acquire new resources from a cloud platform and will select the best type of resource (instance) to lease based on the characteristics of the task it is intended for.

1.4 Objectives

- Study the Cloudbus Workflow Management System thoroughly with emphasis in the following components:
    - Workflow Engine
    - Scheduling and task management modules
    - Cloudbus Broker

- Learn how to configure and run application workflows on the Cloudbus Workflow Management System

- Investigate the pricing model of IaaS cloud providers

- Investigate different workflow scheduling techniques on grid and cloud environments

- Study and understand the Particle Swarm Optimization algorithm

- Model a Particle Swarm Optimization approach that tackles the workflow scheduling problem based on static resources. Propose a high level scheduling heuristic that embeds this model

- Model a Particle Swarm Optimization approach that tackles the workflow scheduling problem based on dynamic resources. Propose a high level scheduling heuristic that embeds this model

- Implementation (due to time constraints only one of the algorithms will be implemented as part of the CWMS, this algorithm will be the static resource approach)

    - Design the integration of the static resource approach with the CWMS

    - Implement the static resource solution as a part of the CWMS

    - Test and debug the algorithm

    - Evaluate the performance of the algorithm and compare it with another scheduling technique

1.5 Motivation

Distributed environments such as grids and clouds provide large scale applications with a powerful platform on which they can be deployed and executed. However, the resources offered by these platforms are generally charged on a pay per use basis and user applications can incur in large costs if not planned properly. Therefore, it is important to have mechanisms that reduce the overall execution cost of these applications.

2 Technical Review

2.1 Cloudbus Workflow Engine Management System

The Cloudbus Workflow Management System (CWMS) [43] is a platform that allows scientist to express their applications as workflows and execute them on distributed resources. In particular, the CWMS enables the creation, monitoring and execution of large scale scientific workflows on distributed environments such as grids and clouds. The main feature of the system is its capability of transparently managing computational processes and data by hiding the orchestration and integration details among the distributed resources [44]. The key components of the system are depicted in figure 4 and the detailed architecture on figure 5.

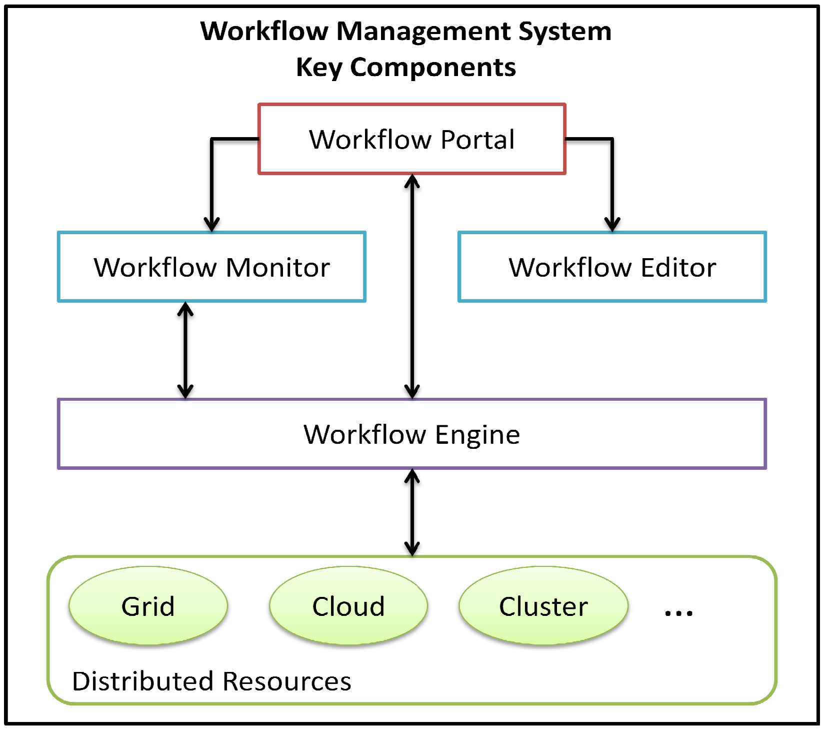


Figure 4 Key components of the CWMS

The Workflow Portal is the entry point to the system. It provides a web based user interface for the users to create, edit, submit and monitor their applications. A workflow editor is embedded in this component, the editor enables user to graphically create new workflows by defining tasks and their dependencies as well as modify existing ones. Additionally, the Workflow Portal provides a submission page that allows users to upload any necessary data and configuration input files needed to run a workflow. Another important feature is the monitoring and output visualization page which allows users to observe the execution progress of multiple workflows and to view the final output of an application. Furthermore, the portal offers users a resource information page which displays the information of all the current available computing resources.

The Workflow Editor is accessed through the portal and it provides a GUI that enables users to create or modify a workflow using drag and drop facilities. The workflow is modeled as a Directed Acyclic Graph (DAG) with nodes and links which represent tasks and dependencies between tasks. Moreover, the editor converts the graphical model designed by the users into an XML based workflow language called xWFL which is the format understood by the underlying workflow engine.

The Workflow Monitor is also accessed through the portal and it provides a GUI for monitoring the status of every task in a specific workflow. For instance, tasks can be on a ready, executing, stage in or completed status, each of which is represented in a different color. Additionally, users have access to information such as the host in which a task is running, the number of jobs being executed (in case of parameter sweep applications) and the failure history of each task. The Workflow Monitor displays and hence relies on the information produced by the Workflow Engine, the interaction between these two components takes place via an event mechanism using tuple spaces. In broad terms, whenever the state of a task changes, the monitor is notified and as a response to the event, it retrieves the new state and any relevant task metadata from a central database.

At the core of the CWMS is the Workflow Engine (WFE). This component interprets a workflow described in xWFL language and schedules the corresponding tasks on the available resources. This project focuses on this component and hence it will be detailed in section 2.1.1.

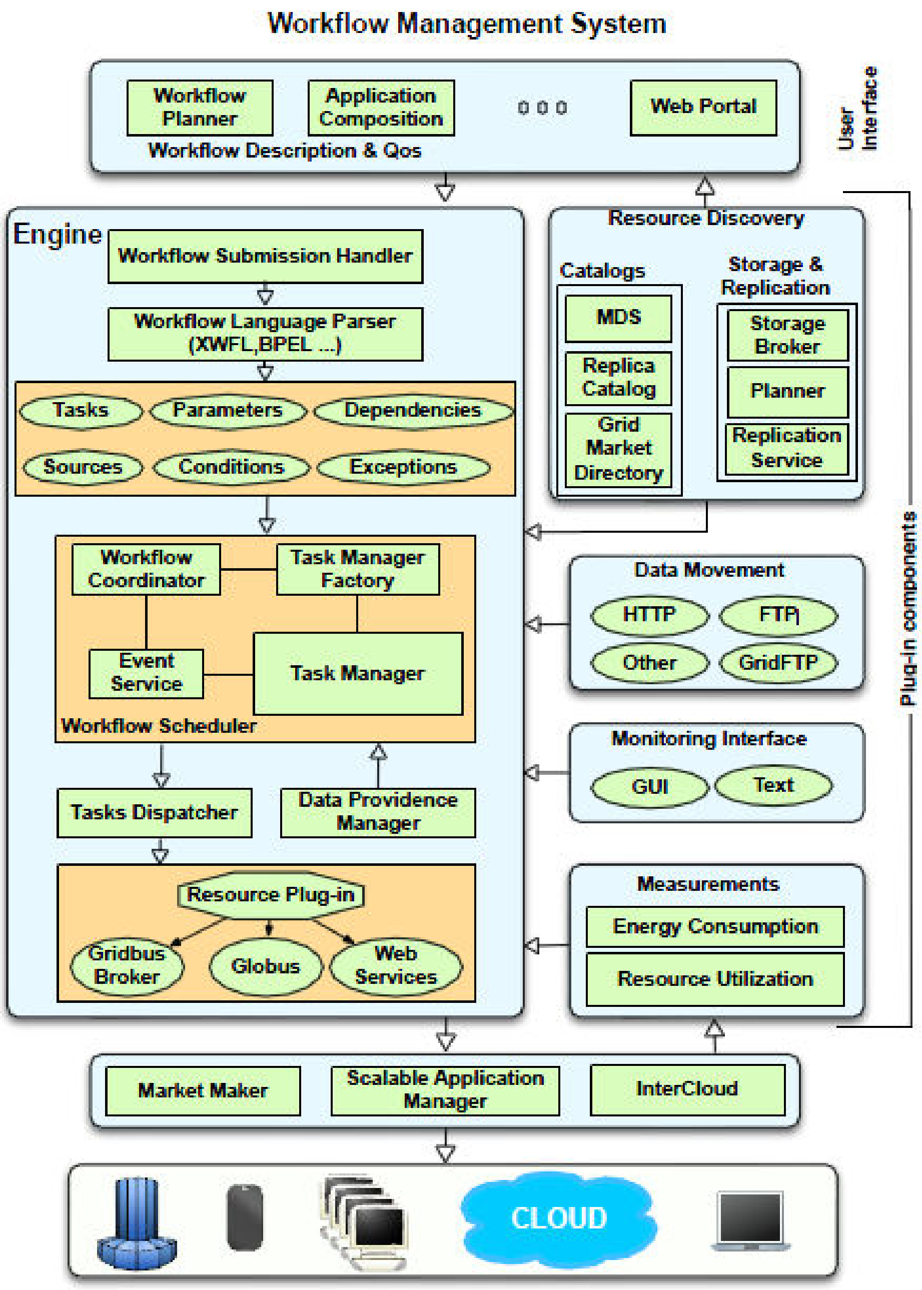


Figure 5 Cloudbus Workflow Management System Architecture [36]

2.1.1 Cloudbus Workflow Engine

The workflow engine is the core of the workflow management system; its main responsibilities include scheduling tasks, dispatching, monitoring and managing their execution on remote resources. As shown in figure 5, the workflow engine has six main subsystems: workflow submission, workflow language parser, resource discovery, dispatcher, data movement and workflow scheduler.

The workflow portal or any other client application submits a workflow for execution to the workflow engine. The submitted workflow must be specified in an xml based language called xWFL. This language enables users to define all the characteristics of a workflow such as tasks and dependencies among others. This xml expressed workflow is then processed and interpreted by a subsystem called the workflow language parser. This subsystem creates objects representing tasks, parameters, data constraints and conditions based on the information contained on the xml file. From this point, these objects will constitute the base of the workflow engine as they are the ones containing all the information regarding the workflow that needs to be executed. Once this information is available, the workflow is scheduled and tasks are mapped to resources based on a specific scheduling policy. After this, the engine uses the Cloudbus Broker as a dispatcher; this component deploys and manages the execution of tasks on the remote resources.

The Cloudbus Broker [45] provides a set of services that enable the interaction of the workflow engine with remote resources. It mediates access to distributed resources by discovering them, deploying and monitoring tasks on specific resources, accessing the required data during task execution and consolidating results. Two additional components that aid in the execution of the workflow are the resource discovery service and the data movement service. The resource discovery service helps in the discovery of suitable resources by querying information services such as the Globus MDS, directory catalogs and replica catalogs. Finally, the data movement component offers services that allow the transfer of data between the engine and remote resources based on protocols such as FTP and GridFTP.

Workflow Scheduling Component

The workflow engine has a decentralized scheduling system that supports just in time planning and allows resource allocation to be determined at run time [43]. Each task has its own scheduler called Task Manager (TM). The TM may implement any scheduling heuristic and is responsible for managing the task processing, resource selection and negotiation, task dispatching and failure handling. At the same time, a Workflow Coordinator (WCO) is responsible for managing the lifetime of every TM as well as the overall workflow execution. Based on this, the architecture offers flexibility in the sense that the scheduling can be done at workflow level by the WCO, task level by each TM or a combination of both, depending on the requirements of the application.

Figure 6 shows the interaction between the different components involved in the scheduling process. The WCO creates and starts a TM based on the task’s dependencies and any other specific scheduling heuristic being used. Each TM has a task monitor which constantly checks the status of the remote task and a pool of available resources to which the task can be assigned. The communication between the WCO and the TMs takes place via events registered in a central event service.

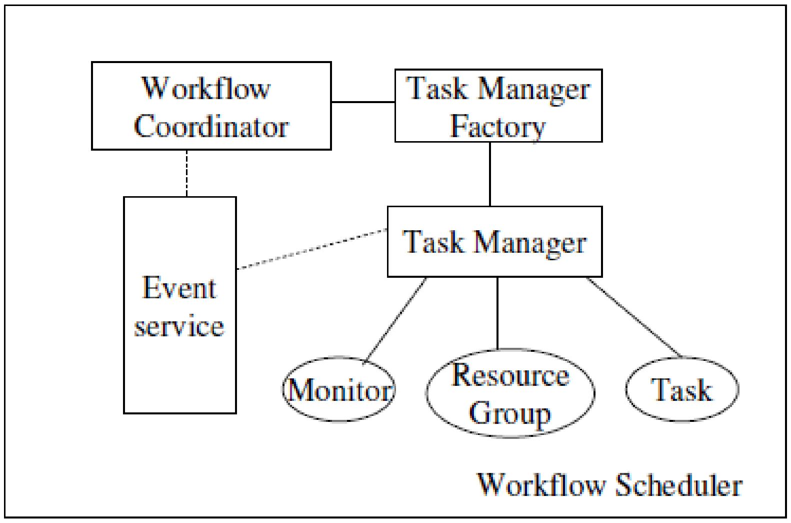


Figure 6 WE Scheduling Architecture

Each TM is independent and may have its own scheduling policy, this means that several task managers may run in parallel. Additionally, the behavior of a TM can be influenced by the status of other task managers. For instance, a task manager may need to put its task execution in hold until its parent task finishes running in order for the required input data to be available. For this reason, TMs need to interact with each other just as the WCO needs to interact with every TM; once again this is achieved through events using a tuple space environment.

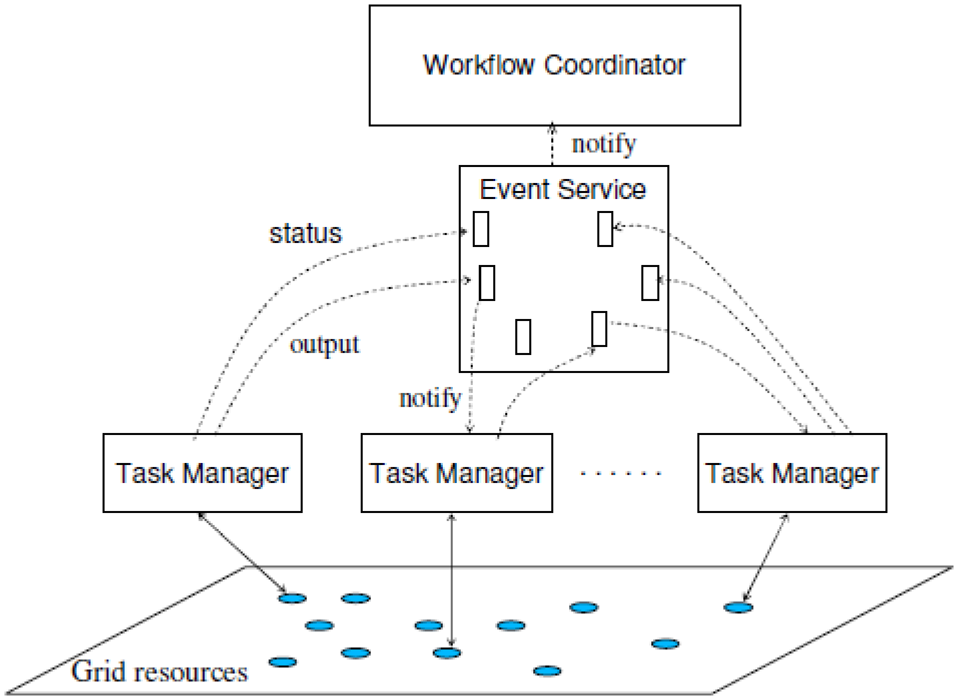


Figure 7 WE Event driven communication [44]

Currently, the workflow engine has a basic scheduling heuristic implemented [35]. First level tasks, which are tasks that have no parents, are the first ones to be scheduled and executed; these are assigned to the first available resources. As parent tasks finish running and produce any necessary output, children tasks become ready for execution. These tasks are then queued and after *polling time* the scheduler assigns them to any available resource to which they get dispatched for execution. The choice of *polling time* is left up to the user and it depends on the number of tasks in the workflow, the scheduling technique, and resource management policies among other factors. The pseudo code for this algorithm is shown in algorithm 1. The contribution of this project is a scheduling policy designed to minimize the total execution cost and evenly utilize the available resources and it is described in subsequent sections.

Algorithm 1 Just In Time Scheduler

```
1. For each root task
    1.1. Assign root task to an available compute resource
2. Repeat until all tasks are scheduled
    2.1. For each task that is ready for execution
        2.1.1. Assign the ready task to any available compute resource
    2.2. Dispatch all the mapped tasks
    2.3. Wait for POLLING_TIME
    2.4. Update the ready task list
```

Algorithm 1 WE Just in Time Scheduling heuristic

2.2 Particle Swarm Optimization

Particle Swarm Optimization is an evolutionary computational technique based on the behavior of animal flocks. It was developed by Eberhart and Kennedy [2] in 1995 and has been widely researched and utilized ever since [2]. The algorithm is a stochastic optimization technique in which the most basic concept is that of particle. A particle simply represents an individual (i.e. fish or bird) which has the ability to move or fly through the defined problem space; based on this, each particle represents a candidate solution to the optimization problem. At a given point in time, the movement of particles is defined by their velocity which is represented as a vector and therefore has magnitude and direction. This velocity is determined by the best position in which the particle has been so far and the best position in which any of the particles has been so far. Based on this, it is imperative to be able to measure how good (or bad) a particle's position is; this is achieved by using a fitness function which measures the quality of the particle's position and varies from problem to problem, depending on the context and requirements.

Each particle is represented by its position and velocity. Additionally, particles keep track of their best position *pbest* and the global best position *gbest*; values that are determined based on the fitness function. The algorithm will then at each step, change the velocity of each particle towards the *pbest* and *gbest* locations. How much the particle moves towards these values is weighted by a random term, with different random numbers generated for acceleration towards *pbest* and *gbest* locations [1]. The algorithm will continue to iterate until a stopping criterion is met; this is generally a specified maximum number of iterations or a predefined fitness value considered to be good enough. On each iteration, the position and velocity of a particle are updated based on equations 1 and 2 respectively. The pseudo code for the algorithm is shown in algorithm 2.

$\vec{x}_i(t+1) = \vec{x}_i(t) + \vec{v}_i(t)$ **Equation 1**

$\vec{v}_i(t+1) = \omega \cdot \vec{v}_i(t) + c_1 r_1\big(\vec{x}_i^*(t) - \vec{x}_i(t)\big) + c_2 r_2(\vec{x}^*(t) - \vec{x}_i(t))$ **Equation 2**

Where:

$\omega = inertia$
$c_i = accelearion\ coefficient, i = 1{,}2$
$r_i = random\ number, i = 1{,}2\ and\ r_i \in [0{,}1]$
$\vec{x}_i^* = best\ position\ of\ particle\ i$
$\vec{x}^* = position\ of\ the\ best\ particle\ in\ the\ population$
$\vec{x}_i = current\ position\ of\ particle\ i$

The velocity equation contains various parameters that affect the performance of the algorithm; moreover, some of them have a significant impact on the convergence of the algorithm. One of these parameters is ω, it is also known as the inertia factor or weight and it is crucial for the algorithm's convergence. This weight determines how much previous velocities will impact the current velocity and defines a tradeoff between the local cognitive component and global social experience of the particles. On one hand, a large inertia weight will make the velocity increase and therefore will favor global exploration. On the other hand, a smaller value would make the particles decelerate and hence favor local exploration. For this reason, a ω value that balances global and local search implies fewer iterations in order for the algorithm to converge.

Conversely, c_1 and c_2 do not have a critical effect in the convergence of PSO. However, tuning them properly may lead to a faster convergence and may prevent the algorithm to get caught in local minima. Parameter c_1 is referred to as the cognitive parameter as $c_1 r_1$defines how much the previous best position matters. On the other hand, c_2 is the social parameter as $c_2 r_2$ determines the behavior of the particle relative to other neighbors.

There are other parameters that are not part of the velocity definition and are used as input to the algorithm. The first one is the number of particles; a larger value generally increases the likelihood of finding the global optimum. This number varies depending on the complexity of the optimization problem but a typical range is between 20 and 40 particles. Other two parameters are the dimension of the particles and the range in which they are allowed to move, these values are solely determined by the nature of the problem being solved and how it is modeled to fit into PSO. Finally, the maximum velocity which defines the maximum change a particle can have in one iteration can also be a parameter to the algorithm; however this value is usually set to be as big as the half of the position range of the particle.

Algorithm 2 Particle Swarm Optimization Algorithm

```
1. Set the dimension of the particles to $\boldsymbol{d}$
2. Initialize the population of particles with random positions and velocities
3. For each particle, calculate its fitness value
4. Compare the particle's fitness value with the particle's $\boldsymbol{pbest}$. If the current value is better
   than $\boldsymbol{pbest}$ then set $\boldsymbol{pbest}$ to the current value and location
5. Compare the particle's fitness value with the global best $\boldsymbol{gbest}$. If the particle's current
   value is better than $\boldsymbol{gbest}$ then set $\boldsymbol{gbest}$ to the current value and location
6. Update the position and velocity of the particle according to equations 1 and 2
7. Repeat from step 3 until the stopping criterion is met.
```

Algorithm 2 PSO

3 Algorithm for Grid Environments

This section describes the proposed algorithm targeting grid environments in which the set of resources that are to be used to execute the workflow tasks is known in advance.

3.1 Definitions and Assumptions

Before presenting the problem formulation it is important to introduce some definitions and assumptions made.

Workflow

A workflow W can be represented as an acyclic directed graph (DAG) in which $W = (T, D)$. In this definition, $T = \{t_1, t_2, \dots, t_n\}$ is the set of tasks that comprise the workflow and $D = \{(t_i, t_j) | \forall\ i, j\ \in [1, n]\ for\ which\ a\ data\ dependency\ exists\ between\ tasks\ t_i\ and\ t_j\}$ corresponds to the set of dependencies between the workflow tasks.

Resources

The set of available heterogeneous resources is defined as $R = \{r_1, r_2, \dots, r_k\}$.

Execution Cost

The execution cost E_{ij} of task t_i in resource r_j is calculated based on the size of the task, the processing capacity and the cost of the resource.

The algorithm proposed makes a series of assumptions in order to calculate E_{ij}:

1. The size I_i of each task (in number of floating point operations FLOP) is known in advance.
2. The processing capacity P_j (in number of floating point operations per second FLOPS) for each resource is known in advance.
3. The cost per unit of time C_j for each available resource is available in advance.

With this information, E_{ij} can be calculated as shown in equation 3.

$E_{ij} = (I_i/P_j) * C_j$ **Equation 3**

3.2 Problem Formulation

Scheduling heuristics may have different objectives, this work focuses on finding a task to resource mapping in such way that the total execution cost of the workflow application is minimized and the available resources are evenly utilized. This means that a balance between the minimum execution cost and an even load distribution among all the resources is desired; ideally all the resources should be utilized.

Based on the previous formal definitions and the stated goals, the scheduling problem addressed in this section can be stated as follows:

Assign each task in T to a resource in R such that each resource gets a similar number of tasks and the total cost $E = \sum_{i=1}^{n} E_{ij}$ for executing the workflow is minimized.

3.3 Workflow Scheduling based on PSO

There are two key steps when modeling a particle swarm optimization problem. The first one is defining how the problem will be encoded, that is defining how the solution will be represented. The second one is defining how the goodness of a particle will be measured, that is defining the fitness function.

The first representation that needs to be established is the meaning and the dimension of a particle. For the scheduling scenario presented here, one particle would represent the set of tasks that need to be allocated; hence, the dimension of a particle would be equal to the number of tasks that need to be assigned to a resource. Based on this, each dimension of a particle would represent a compute resource assigned to the task represented by that particular dimension. In particular, the value assigned to each dimension of a particle corresponds to the computing resources index. Therefore, the particle represents a mapping of a resource to a task. A sample particle and its position are depicted in figure 8. Another factor that needs to be determined is the range in which a particle is allowed to move, because the position represents compute resources then this range is defined to be between one and the number of available resources.

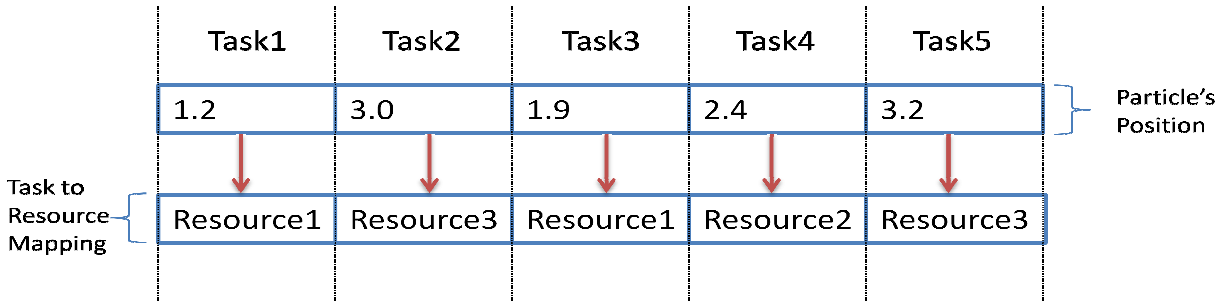


Figure 8 Encoding of a particle's position

Since the fitness function is used to determine how good a potential solution is, it needs to reflect the objectives of the scheduling problem. In this case, we want to minimize the execution cost while distributing the tasks evenly on the resources. Based on this, the fitness function will be minimized and it will have two components; the first one will represent the execution cost and the second one will represent how evenly the resources are being used.

The first component is straightforward; the execution cost of a mapping is simply the sum of the cost of each task on its assigned resource. If the fitness function was defined as this single component then the algorithm would tend to assign every task to the cheapest resource. To avoid this, a second component is added. When calculating the fitness function for a particular mapping, if a task is assigned to a resource that already had a task assigned to it then a penalization value is added to the fitness value. This grows proportionally to the number of tasks a resource has assigned to it. This is depicted in the fitness function presented in equation 4. The function takes as input a particle's position, which is an array of size d where d is the dimension of the particle and in this particular scenario, the number of tasks that need to be mapped onto a resource.

$$g(position) = \sum_{i=0}^{d}(\lambda(E_{i,position[i]}) + (1-\lambda)(n_{position[i]} * \varepsilon)) \quad \textbf{Equation 4}$$

Where:

$n_j = number\ of\ tasks\ assigned\ to\ the\ resource\ j$
$\varepsilon = penalization\ value\ for\ assigning\ a\ task\ to\ a\ previously\ used\ resource$
$\lambda = value\ between\ 0\ and\ 1\ that\ gives\ more\ weight\ to\ either\ the\ load$
$\quad balancing\ component\ or\ the\ cost\ minimization\ one$

In the approach presented here, ε was taken to be the average computation cost of each task on each resource multiplied by 10. The reasoning behind this is that this number is a value that increases the fitness value by a high enough amount so that the algorithm also consider options that are more costly but on which resources are better utilized.

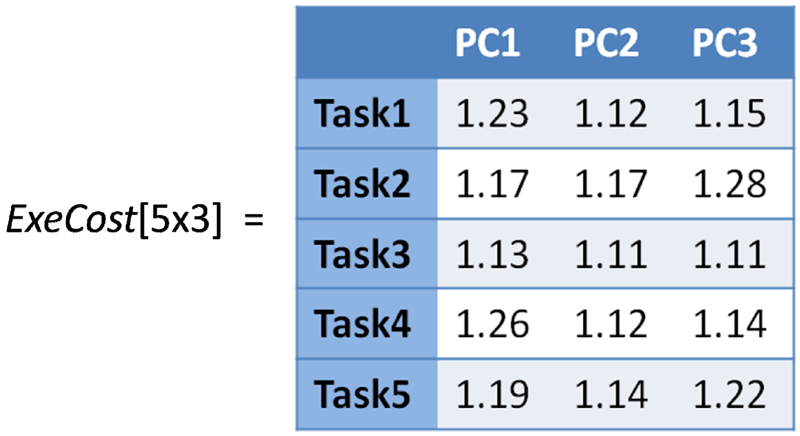
Having modeled the PSO problem, a higher level scheduling heuristic that embeds the PSO algorithm is needed. The pseudo code for this heuristic is shown in algorithm 3. The first step is to estimate the execution cost of every workflow task on every resource. This can be expressed as a matrix in which the rows represent the tasks, the columns the resources and the entry $ExeCost[i,j]$ contains the cost of executing task i in resource j. This cost is calculated using equation 4 and it is the basis for calculating the fitness value in the PSO algorithm. A sample matrix is illustrated in figure 9. The second step consists in getting the list of all the tasks that are ready for execution; these tasks are those which have no incoming dependencies (i.e. first level tasks) or child tasks whose parent or parents finished executing and hence have the necessary input available. In the third step, the compute resources available are retrieved. At this point, the information required to execute the PSO algorithm is available and hence the task to resource mapping is computed in step four. In step 5 every task is assigned to the corresponding resource and dispatched so that its execution begins. After this, the scheduler waits for a predefined amount of time before checking the status of executing tasks and updating the list of tasks ready for execution. This $polling\_time$ can be defined by the user as its optimum value may be influenced by the number of tasks and topology of the workflow. The list of ready tasks is updated based on the tasks completed so far, this enables the algorithm to execute the workflow tasks on a specific order which is dictated by the dependencies defined between them. Finally, the algorithm loops back to step two and iterates until all the tasks have been scheduled and dispatched.

Algorithm 3 Scheduling Heuristic

```
1.  Compute the execution cost of each task on every resource according to equation 4
2.  Get the set of tasks ready for execution $\boldsymbol{T_r}$
3.  Get the set of available resources $\boldsymbol{R}$
4.  Get the task to resource mapping by running $\boldsymbol{PSO(T_r, R)}$
5.  For each task $\boldsymbol{t}$ in $\boldsymbol{T_r}$
    5.1. Assign $\boldsymbol{t}$ to $\boldsymbol{r_i} \in \boldsymbol{R}$ based on the mapping produced in step (4)
    5.2. Dispatch $\boldsymbol{t}$ for execution
```

```
6. Wait for $polling\_time$
7. Update $T_r$ with new tasks ready for execution
8. Repeat from step 5 until there are no more tasks to be scheduled
```

Algorithm 3 Scheduling heuristic for grid environments



ExeCost[5x3] =

	PC1	PC2	PC3
Task1	1.23	1.12	1.15
Task2	1.17	1.17	1.28
Task3	1.13	1.11	1.11
Task4	1.26	1.12	1.14
Task5	1.19	1.14	1.22

Figure 9 Cost execution matrix

3.4 Implementation

The cost minimization algorithm was implemented and integrated into the Cloudbus Workflow Management System. In particular, several extensions and changes were made to the workflow engine component in order to add the new scheduling policy. However, the overall architecture and interaction between components remains the same.

In order to execute applications, the workflow engine requires three xml files to be provided. The first one is the application file and it specifies the workflow tasks and dependencies. The second is the service file and it describes the resources available for processing tasks. The third one is the credentials file and it defines the security credentials needed to access the resources. These files are parsed into objects that are used later on used in the scheduling and execution processes.

For the scheduling component, only the application and service files are relevant. The schemas and parsers for these files were extended in order to provide the extra information required by the PSO algorithm. The application file was modified so that the number of floating point operations (FLOP) for each task was included in the task definition. The service file was updated to include the processing capacity in floating point operations per second (FLOPS) in the compute resource definition. These changes are illustrated in figures 10 and 11. The parsers for both xml files were updated to support the new schemas. Additional to this, the WfTask and

ComputeServer classes which represent a workflow task and compute resource respectively, were extended to support the new properties.

Application File

Old Version

```
<task name="task1" paramsweep="false">
    <executable>
        <name value="test" I_OModel="many_many"/>
        <service serviceID="service1"/>
        <input>
            <port number="0" type="file" value="in.out"/>
        </input>
        <output>
            <port number="1" type="file" value="out.out"/>
        </output>
    </executable>
</task>
```

New Version

```
<task name="task1" paramsweep="false">
    <mflop>10000</mflop>
    <executable>
        <name value="test" I_OModel="many_many"/>
        <service serviceID="service1"/>
        <input>
            <port number="0" type="file" value="in.out"/>
        </input>
        <output>
            <port number="1" type="file" value="out.out"/>
        </output>
    </executable>
</task>
```

Figure 10 Changes made to the application file

Service File

Old Version

```
<service type="compute" cost="0.08" mappingID="resource1">
    <compute middleware="fork">
        <fork hostname="belle.csse.unimelb.edu.au"/>
    </compute>
</service>
```

New Version

```
<service type="compute" cost="0.08" mappingID="resource1">
    <compute middleware="fork" mflops="4400">
        <fork hostname="belle.csse.unimelb.edu.au"/>
    </compute>
</service>
```

Figure 11 Changes made to the service file

The workflow engine's architecture allows for scheduling decisions to be made base on either local or global information. The classes that make this possible are the WorkflowCoordinator and TaskManager, these are the core of the scheduling component. On one hand, the workflow coordinator has access to the information of all tasks and hence is responsible for managing the high level scheduling of the workflow. On the other hand, each task is assigned to a task manager which means that it only has access to the information of that task and is responsible for the task level or local scheduling. The heuristic depicted in algorithm 3 uses the information of the entire workflow to schedule the tasks and therefore is implemented in the WorkflowCoordinator class. The class hierarchy that supports the implementation of algorithm 3 is explained in the class diagram depicted in figure 12 and the sequence diagram in figure 13.

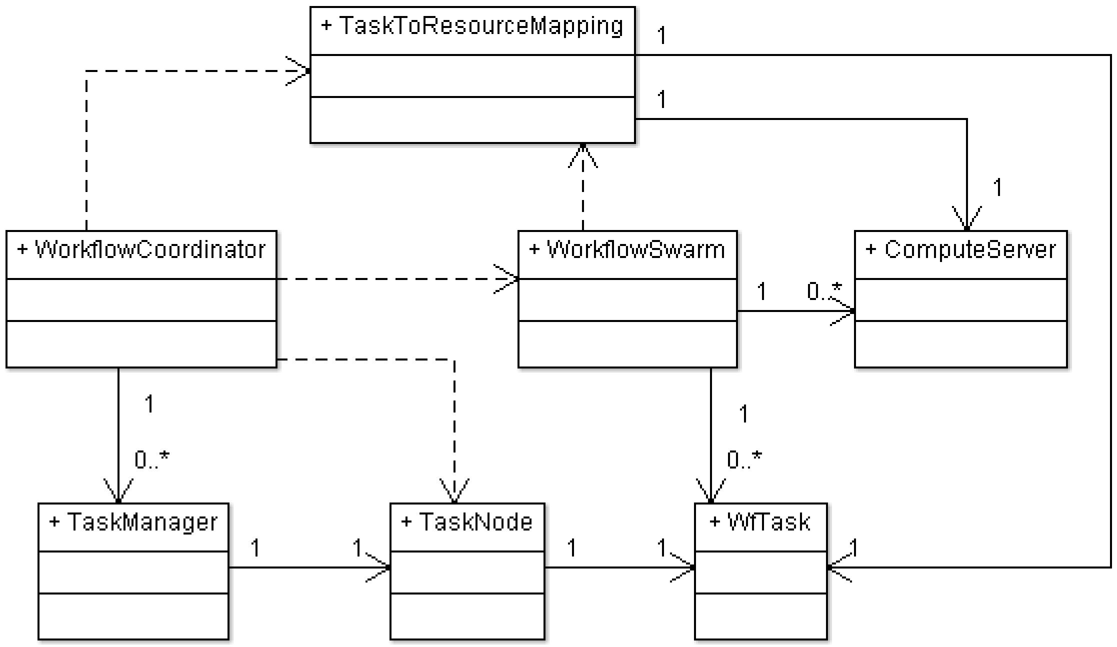


Figure 12 High level class diagram for the grid heuristic

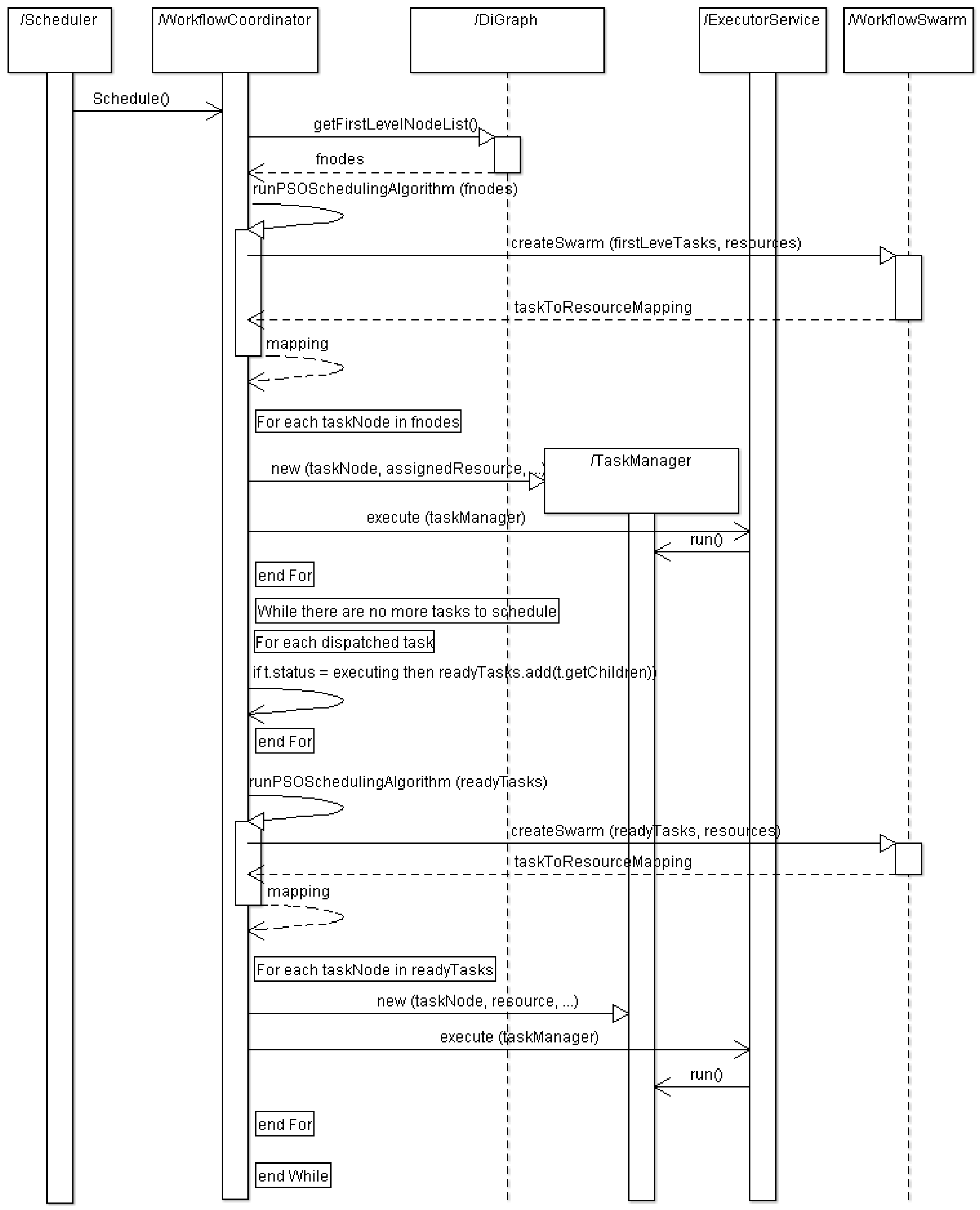

/Scheduler
/WorkflowCoordinator
/DiGraph
/ExecutorService
/WorkflowSwarm
Schedule()
getFirstLevelNodeList()
fnodes
runPSOSchedulingAlgorithm (fnodes)
createSwarm (firstLeveTasks, resources)
taskToResourceMapping
mapping
For each taskNode in fnodes
/TaskManager
new (taskNode, assignedResource, ...)
execute (taskManager)
run()
end For
While there are no more tasks to schedule
For each dispatched task
if t.status = executing then readyTasks.add(t.getChildren())
end For
runPSOSchedulingAlgorithm (readyTasks)
createSwarm (readyTasks, resources)
taskToResourceMapping
mapping
For each taskNode in readyTasks
new (taskNode, resource, ...)
execute (taskManager)
run()
end For
end While

Figure 13 Sequence diagram for the grid heuristic

The PSO algorithm was implemented as a separate component. It was developed using the Java library JSwarm-PSO [27]. The algorithm takes as input two lists, one of WfTask objects representing the tasks to be scheduled and another of ComputeServer objects representing the resources available. As a first step, all the parameters required by PSO are initialized. The particle's dimension is set to be the number of input tasks. The position range in which the particle can move is set to between zero and the number of resources minus one; this creates an implicit mapping between the index of the resources and the particle's position. Other parameters such as c_1 , c_2 and ω are also initialized in this stage. The required fitness function that the algorithm will use is implemented in the ComputeFitness class and it is based on equation 4. The JSarm-PSO library is used to execute the core logic of the algorithm and after it finishes, the output is interpreted and translated into a resource to task mapping usable by the workflow engine. In particular, the output of the algorithm is a list of TaskToResourceMapping objects. A TaskToResourceMapping object has two properties, the first one is of type Wftask and the second one of ComputeServer. The class diagram for the PSO component is depicted in figure 14.

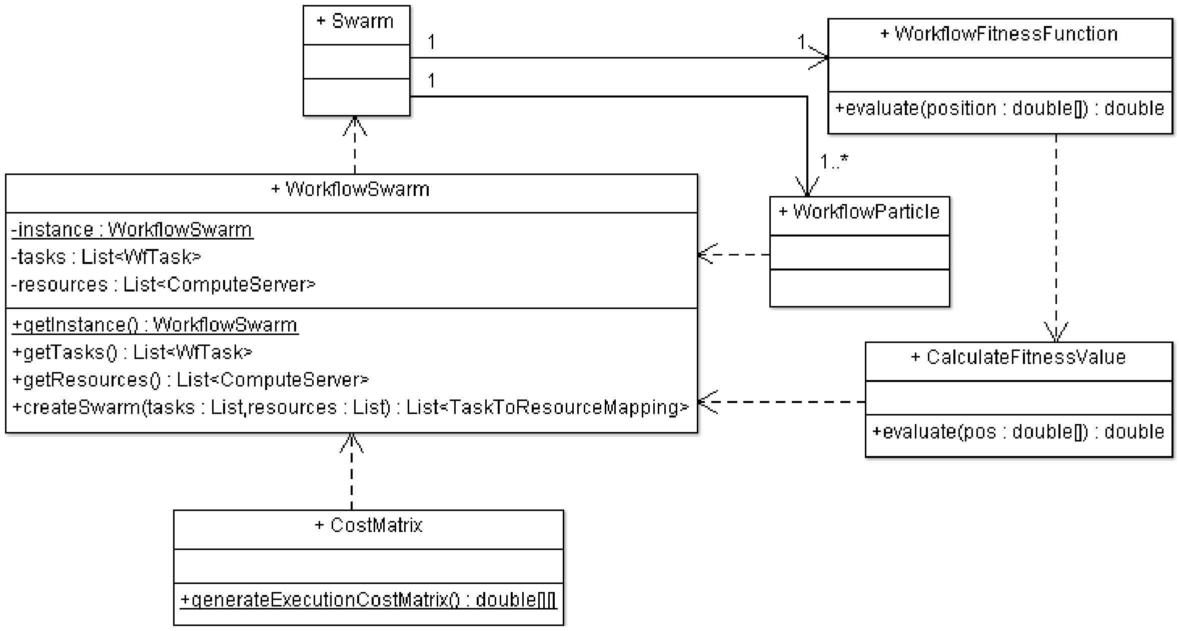

+ Swarm
1
1
1
+ WorkflowFitnessFunction
+evaluate(position : double[]) : double
1..*
+ WorkflowSwarm
-instance : WorkflowSwarm
-tasks : List<WfTask>
-resources : List<ComputeServer>
+getInstance() : WorkflowSwarm
+getTasks() : List<WfTask>
+getResources() : List<ComputeServer>
+createSwarm(tasks : List,resources : List) : List<TaskToResourceMapping>
+ WorkflowParticle
+ CalculateFitnessValue
+evaluate(pos : double[]) : double
+ CostMatrix
+generateExecutionCostMatrix() : double[][]

Figure 14 PSO class diagram

3.5 Experiments and Results

The proposed heuristic was contrasted with two different approaches. The first one is a round robin algorithm, a simple approach in which each task is scheduled in the next available resource. When comparing the PSO approach with this one, the total workflow execution cost was used as a metric. To evaluate the load balance feature of the algorithm, a simpler version of it in which the fitness function comprises only of the execution cost was implemented and used to evaluate the algorithm. Additionally, experiments with different values of the PSO parameters where held.

The first set of experiments carried out were those that varied the different PSO parameters, this was done in order to find the best configuration for the given problem. These experiments were held with a simulated workflow of 20 heterogeneous tasks and 10 heterogeneous resources; each experiment was repeated 10 times and the values shown are an average of these results. Different values for the number of particles, c_1, c_2 and ω were tried and the results are shown in tables 2, 3 and 4.The best configuration found was for 25 particles, , $c_1 = 0.9$, $c_2 = 0.9$ and $\omega = 0.95$. The rest of the experiments held were done using these parameter values.

particles	c1	c2	w	cost
25	0.9	0.9	0.95	22.68809
25	1.4	1.4	0.95	23.13056
25	1.9	1.9	0.95	23.18673
25	2.4	2.4	0.95	22.93031
25	2	1	0.95	22.78724
25	1	2	0.95	22.94646
25	0.5	1.5	0.95	23.00431
25	1.5	0.5	0.95	23.04111

Table 2 PSO results for different values of $c_1$and c_2 parameters

particles	c1	c2	w	cost
25	0.9	0.9	0.95	22.68809
25	0.9	0.9	1	22.88573
25	0.9	0.9	1.05	23.0757
25	0.9	0.9	1.1	23.0146
25	0.9	0.9	2	23.26383
25	0.9	0.9	3	23.4639
25	0.9	0.9	4	23.45969

Table 3 PSO results for different ω values

particles	c1	c2	w	cost
5	0.9	0.9	0.95	23.38486
10	0.9	0.9	0.95	23.18674
15	0.9	0.9	0.95	22.95487
20	0.9	0.9	0.95	22.7174
25	0.9	0.9	0.95	22.89023
30	0.9	0.9	0.95	22.82647

Table 4 PSO results for different number of particles

An additional set of experiments, comparing the proposed PSO approach with a round robin one were performed. These tests were done using the workflow engine mentioned before. Eight different workflows with the number of tasks ranging from 5 to 20 were used; the tasks had varied sizes ranging from 10 to 50000 MFLOP. Five different resources were used to schedule the workflows; each compute resource had different processing capacities in terms of MFLOPS and different costs per hour. Figures 15 and 16 show the achieved results. On average, by using PSO, the overall workflow execution cost was reduced by 28.5%.

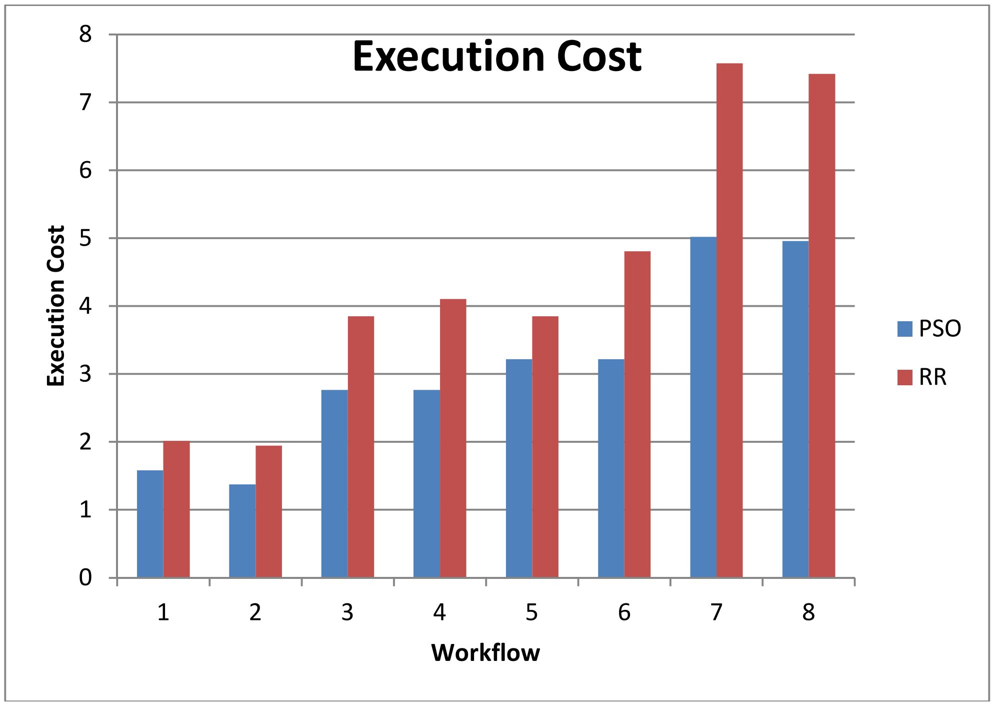


Figure 15 Execution cost of 8 workflows using PSO and Round Robin

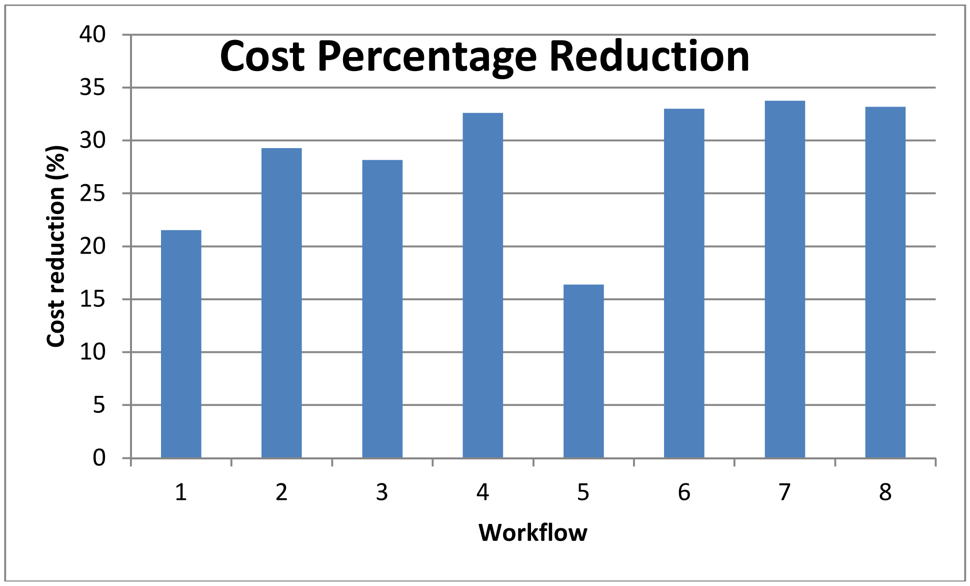


Figure 16 Percentage in which PSO reduces the Round Robin execution

To test the load balancing mechanism experiments were held with a version of the algorithm in which only the execution cost was part of the fitness function. The results show that the proposed heuristic greatly improves the distribution of the tasks over the different reductions. Results are shown in table 5 and figure 17.

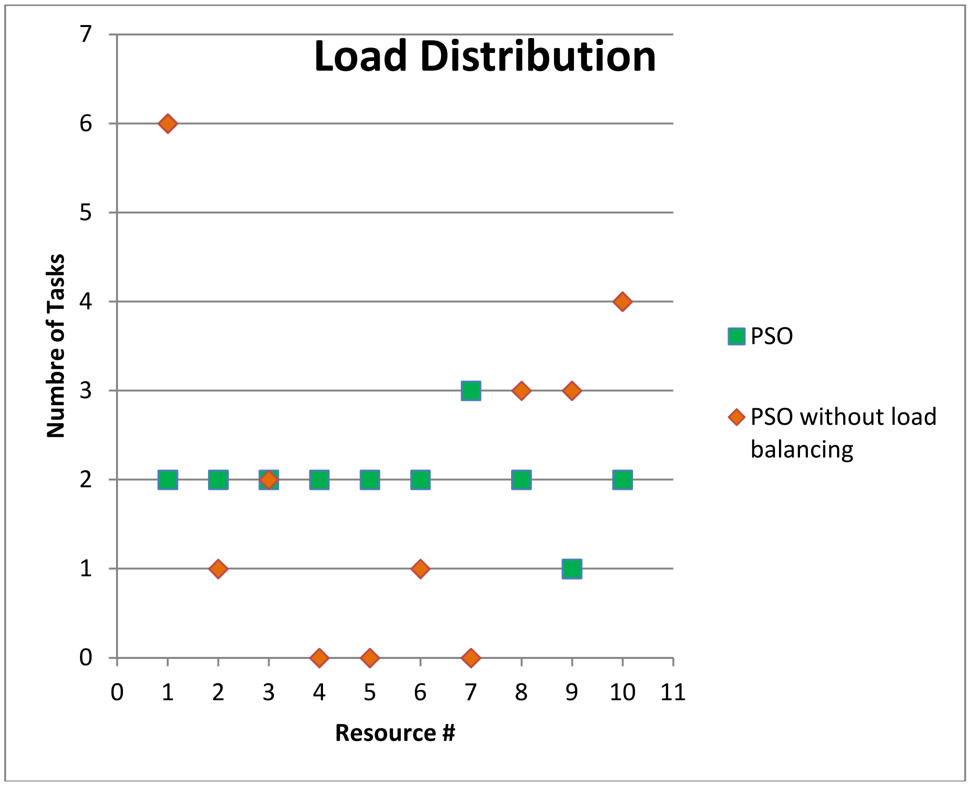


Figure 17 Number of tasks per resource

	PSO	PSO without load balancing
	20 tasks - 10 resources	
Standard Deviation	0.447214	1.897366596
Variance	0.2	3.6
	20 tasks - 5 resources	
Standard Deviation	0	0.894427191
Variance	0	0.8
	20 tasks - 20 resources	
Standard Deviation	0.547723	1.264911064
Variance	0.3	1.6

Table 5 Variance and standard deviation of the load distribution

Even though the performance of PSO greatly depends on the value of the input parameters, the experiments held show that the final outcome of the algorithm was not greatly impacted by changing these values (for the range of values tested). However, for larger scale problems, the difference between two set of parameters can mean a considerable cost reduction. The reason for the results not changing much from one configuration to the other might be the scale of the problem, the tests were held with 20 tasks and 10 resources which had an hourly cost between 1.1 and 1.3. If the number of resources and tasks was higher and the difference in machine costs was higher, the search space would be sparser and probable a slight change in the parameter values could greatly impact the result; however, this was considered a reasonable setting for the studied problem.

Regarding the cost minimization strategy proposed, only the processing time of each task on a resource was considered when calculating the workflow cost. Even though that is one of the components of the total execution cost, other aspects such as the data transmission cost should also be considered in order for the scheduling heuristic to be as accurate as possible.

4 Dynamic Resource Leasing Approach

Traditional distributed systems such as clusters and grids are mostly dedicated and static; however, IaaS clouds offer elasticity as the number of resources used can grow and shrink depending on the applications' needs, furthermore, different type of resources are available in terms of memory, number of cores, etc. and these resources are leased on a pay per use basis. Unlike more traditional systems, in the cloud, resources are highly dynamic and heterogeneous and this represents a further challenge when scheduling workflows on these infrastructures. The resource leasing model offered by cloud providers needs to be considered when scheduling jobs. For instance, the number of resources to be leased needs to be determined and the fact that different tasks can be assigned to different types of virtual machines needs to be considered.

Before the advent of cloud computing, distributed environments consisted mostly of dedicated private resources or community grids on which the user application was not charged by using the resources; in these systems the goal was to execute the application as fast as possible and hence, scheduling algorithms were developed to consider only the execution time. The main objective of this algorithm is to develop a set of heuristics that consider the elastic cloud leasing model in order to decrease the cost of executing scientific workflows on an IaaS cloud provider. IaaS platforms offer various types of virtualized resources with different characteristics and cost. These resources are provided in the form of virtual machines (VMs); theoretically, users have access to an infinite number of VMs. These VMs can be leased and used for as long as they are needed and released when they're not required anymore. This gives the users the flexibility to configure their environment according to their specific needs by choosing several features such as the number and types of virtual machine to lease and the period of time for which they will be acquired.

Most research until now has focused on different scheduling problems and techniques that require a set of machines and their information to be available previous to the scheduling process. The algorithm presented in this section is specifically tailored for dynamic environments based on cloud IaaS providers. Instead of assuming that an initial set of compute resources is available, it dynamically selects the best type of instances to lease based on the characteristics of the workflow tasks and the instances offered by the cloud provider.

The algorithm requires certain information to be available in order for it to make appropriate decisions. First, a description of the instances offered by the cloud provided is required; this description must contain at least the name of the instance as recognized by the cloud environment, its cost per hour and its processing capacity in terms of FLOPS. Second, every task in the workflow needs to be described, among other things, in terms of its size specified as number of FLOPs.

The basis of the algorithm is the instance type selection heuristic. This heuristic is based on a single task and a set of instances of different types; the aim is to determine the instance type which will lead to the minimum execution cost of the task. To achieve this, the cost of the task on each instance type is calculated; this cost is calculated based on the time the task will take to execute in the particular instance and the cost per hour of the instance. Algorithm 4 shows how the cost of executing a task in an instance type is calculated.

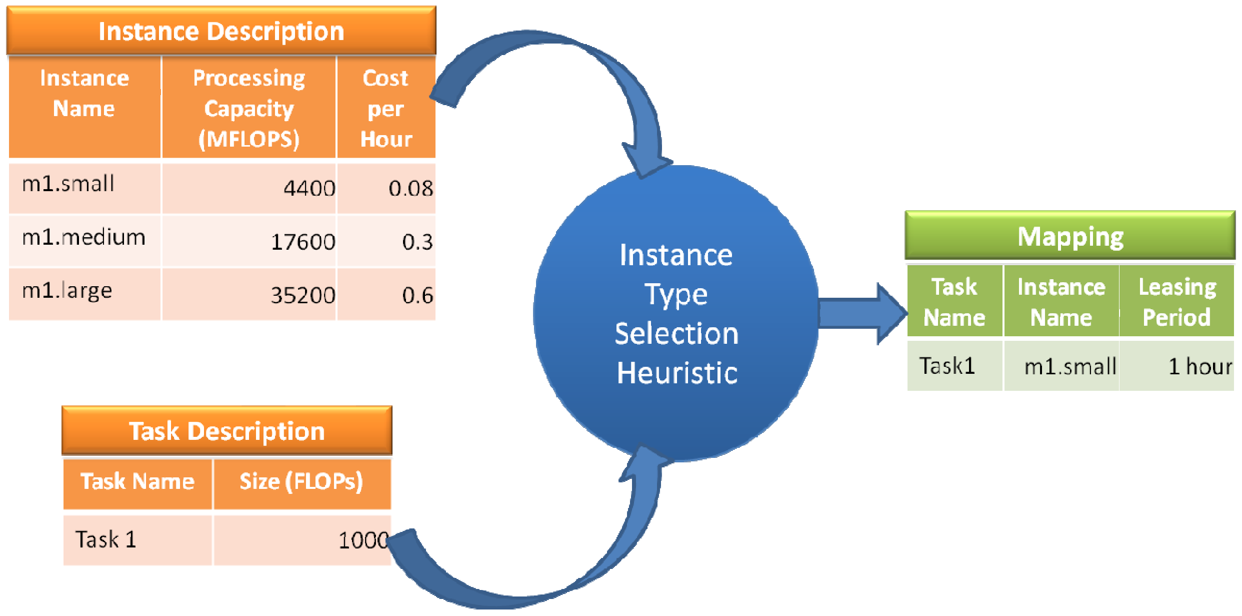


Figure 18 Instance Type Selection Heuristic

Algorithm 4 Computation of $cost_{t,i}$ (Cost of executing task t in instance type i)

```
1. time := t.size / i.processingCapacity
```

```
2.  if time <= 3600 then $\boldsymbol{cost}_{t,i}$:= i.costPerHour
        else $\boldsymbol{cost}_{t,i}$:= Math.ceil(time/3600.0) * i.costPerHour();
```

Algorithm 4 Computation of the cost of executing task t in instance type i

The instance type selection heuristic defines which is the most appropriate instance type for a particular task; however, this does not solve the stated scheduling problem and hence it needs to be embedded into a more robust algorithm so that given a set of tasks, the most appropriate set (i.e. that which minimizes the total execution cost) of instance types is selected. Since PSO is an optimization technique that can be adapted to several problems; this algorithm was used to solve the task to instance type mapping problem.

The encoding of the problem is as follows. The dimension of a particle is the number of tasks in the workflow, each dimension of represents the instance type assigned to the task represented by that particular dimension. Based on this, the particle represents a mapping of an instance type to a task. A sample particle and its position are depicted in figure 19. The range in which a particle is allowed to move is defined to be between one and the number of different instance types available.

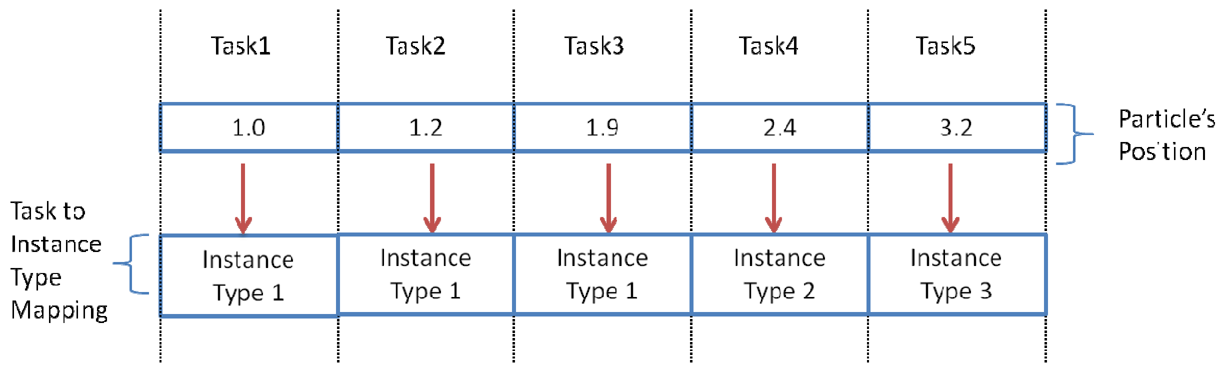


Figure 19 Encoding of a particle

The fitness function is used to measure how good a potential solution or particle is. In this particular case we want to minimize the total execution cost and therefore the fitness value is calculated as the sum of the execution cost of each task on its assigned resource; a cost of a task in a particular resource is computed based on algorithm 4. The function presented in equation 5; it takes as input a particle's position, which is an array of size d where d is the dimension of the particle and in this particular scenario, the number of tasks in the workflow.

$g(position) = \sum_{i=0}^{d} Cost_{t_i, position[i]}$ **Equation 5**

Where

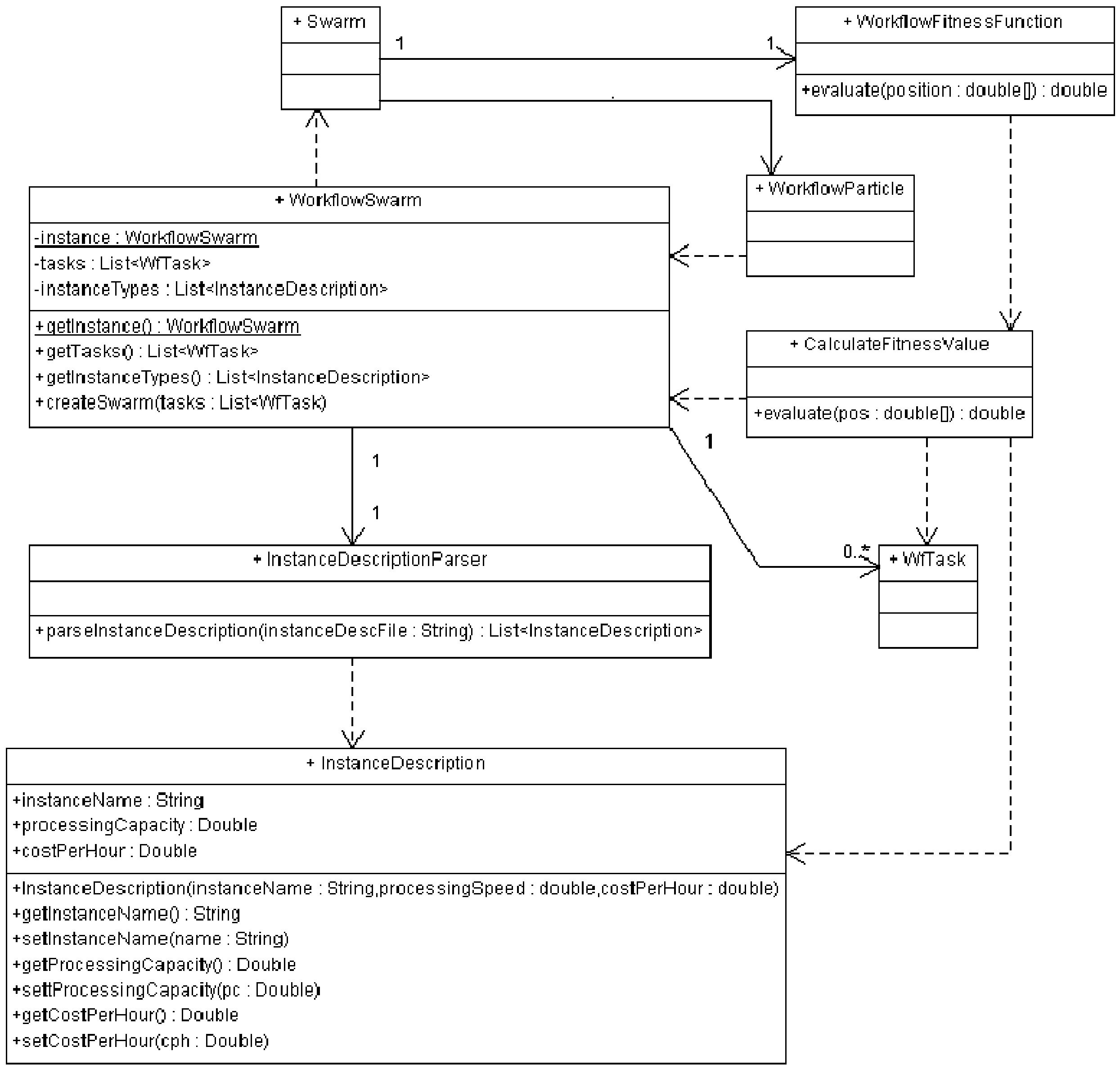
$position[i] \ \in [0, numInstances - 1] \ and\ represents\ the\ index\ of\ an\ instance\ type$

The pseudo code for the algorithm is depicted in algorithm 5.

Algorithm 5 Cloud Cost Minimization Scheduling Heuristic

```
1. Let $T :=$ the set of all tasks in the workflow
2. Let $I :=$ the set of descriptions of all the available instance types
3. Run PSO$(T, I)$
    3.1. Set the particle dimension equal to the number of tasks in the workflow
    3.2. Initialize all particle's position and velocity randomly
    3.3. For each particle, calculate its fitness value based on equation 5
    3.4. Compare the particle's fitness value with the particle's $pbest$. If the current value is
         better than $pbest$ then set $pbest$ to the current value and location
    3.5. Compare the particle's fitness value with the global best $gbest$. If the particle's
         current value is better than $gbest$ then set $gbest$ to the current value and location
    3.6. Update the position and velocity of the particle according to equations 1 and 2
    3.7. Repeat from step 3.3 until the stopping criterion is met.
4. Lease the required instances based on the output of step 3
5. For each $t$ in $T$ which is ready for execution
    5.1. Assign $t$ to resource $r_i$ which corresponds to the type assigned to the task in step 3
    5.2. Dispatch $t$ for execution
6. Wait for $polling\_time$
7. Update $T$ with new tasks ready for execution
8. Repeat from step 5 until there are no more tasks to be scheduled
```

Algorithm 5 Cloud Cost Minimization Scheduling Heuristic



+ Swarm
1
1
+ WorkflowFitnessFunction
+evaluate(position : double[]) : double
+ WorkflowParticle
+ WorkflowSwarm
-instance : WorkflowSwarm
-tasks : List<WfTask>
-instanceTypes : List<InstanceDescription>
+getInstance() : WorkflowSwarm
+getTasks() : List<WfTask>
+getInstanceTypes() : List<InstanceDescription>
+createSwarm(tasks : List<WfTask>)
+ CalculateFitnessValue
+evaluate(pos : double[]) : double
1
1
1
0..*
+ WfTask
+ InstanceDescriptionParser
+parseInstanceDescription(instanceDescFile : String) : List<InstanceDescription>
+ InstanceDescription
+instanceName : String
+processingCapacity : Double
+costPerHour : Double
+InstanceDescription(instanceName : String,processingSpeed : double,costPerHour : double)
+getInstanceName() : String
+setInstanceName(name : String)
+getProcessingCapacity() : Double
+settProcessingCapacity(pc : Double)
+getCostPerHour() : Double
+setCostPerHour(cph : Double)

Figure 20 Class diagram for the cloud algorithm

A simplified version of the algorithm which includes steps 1, 2 and 3 was implemented in order to evaluate its efficiency. The class diagram is depicted in figure 20. The evaluation was done by using the Amazon EC2 instances and pricing policy. Table 6 shows the description of the resource types used for the simulation. As mentioned earlier, the algorithm requires the processing capacity of each instance to be specified in terms of MFLOPS. To identify the MFLOPS each used instance is capable of processing, the work in [46] was used. According to the authors, at peak performance, one ECU (Amazon Compute Unit) equals 4.4 gigaflops per second (GFLOPS); this data is based on Amazon's ECU definition. An ECU is defined to have the equivalent CPU power of a 1.0-1.2 GHz 2007 Opteron or Xeon processor which can perform 4 flops per cycle at full pipeline.

Instance Type	ECUs	Cost per Hour	MFLOPS	Instance Description
m1.small	1	0.08	4400	Standard small
m1.large	4	0.3	17600	Standard large
m1.extraLarge	8	0.6	35200	Standard extra large
c1.medium	5	0.17	22000	High CPU medium
c1.large	20	0.8	88000	High CPU large

Table 6 Instance types offered by Amazon EC2

A simple experiment with two tasks that illustrates how the algorithm works was done. Tables 8 and 9 show the execution time and cost details for two different tasks on each type of instance. Task 1 consists of 10000000 MFLOPs whereas task 2 consists of 100000000. From table 8 it is clear that the instance type on which task 1 would cost the least to execute is m1.small, equivalently, table 9 shows that the best instance type for task 2 is c1.medium. Figure 10 shows the output produced when the proposed algorithm was ran for tasks 1 and 2 and the instance types depicted in figure 7. The mapping created is as expected as the results match those depicted in tables 8 and 9 and the chosen instances are those which minimize the total execution cost.

Task Name	MFLOPs
Task 1	10000000
Task 2	100000000

Table 7 Task 1 and Task 2 description

Task 1					
	m1.small	m1.large	m1.extraLarge	c1.medium	c1.large
execution time (hrs)	0.631313131	0.157828283	0.078914141	0.126262626	0.031565657
cost	0.08	0.3	0.6	0.17	0.8

Table 8 Execution time and cost details of task **1**

Task 2					
	m1.small	m1.large	m1.extraLarge	c1.medium	c1.large
execution time (hrs)	6.313131313	1.578282828	0.789141414	1.262626263	0.315656566
cost	0.56	0.6	0.6	0.34	0.8

Table 9 Execution time and cost details of task 2

Task	Instance Type	Cost
Task 1	m1.small	0.08
Task 2	c1.medium	0.34
Total		**0.42**

Table 10 Output of the cloud algorithm for two tasks

Additional experiments were held and in general, the algorithm maps tasks to instance types which minimize the cost. The output of an experiment ran with 5 tasks of varying sizes and 3 instance types (m1.small, m1.large, m1.extraLarge) is depicted in table 11. The integration of the algorithm and further experiments and evaluation are left as future work.

Task	MFLOPs	Instance Type
Task 1	10000000	m1.small
Task 2	20000000	m1.small
Task 3	30000000	m1.small
Task 4	40000000	m1.medium
Task 5	50000000	m1.medium

Table 11 Output of the cloud algorithm for five tasks

5 Conclusion and Future Work

This project presented two PSO based heuristics for scheduling application workflows on heterogeneous distributed environments in such way that the total execution cost is minimized. The main difference between the two approaches lies on the environment which provides the required computing resources. The first one, referred to as the grid approach, assumes a set of finite resources is available before the scheduling process begins. The second one, referred to as the cloud approach, assumes there are no resources available when the scheduling process begins and instead, machines should be dynamically selected and acquired from a potentially infinite set of heterogeneous resources.

The algorithm for grid environments was implemented and integrated into a workflow management system called CWMS. This solution focused on two objectives, the first one was minimizing the overall execution cost and the second one was balancing the number of tasks assigned to each resource. The results achieved demonstrate the efficacy of the algorithm; compared to a round robin based approach, it decreased the cost in 28% and at the same time distributed the tasks more evenly among the available resources. As future work, the multi objective nature of the problem (minimize cost and load balancing) could be addressed with more sophisticated evolutionary multi objective optimization techniques such as criterion and dominance based

The cloud algorithm was not implemented as part of the CWMS; however, a simplified implementation capable of simulating the main steps of the algorithm was made. Results show that in most of the cases, for a given task, the algorithm selects the instance type that leads to the task's minimum execution cost and hence, the set of selected instances is optimal in terms of cost minimization. It is left as future work to extend the CWMS so that cloud environments are supported and integrate the algorithm to the system. Additionally, more robust experiments and evaluation are required to properly analyze the performance of the algorithm.

An important aspect that contributes to the total execution cost of a workflow in a set of distributed resources is the cost of transferring data from one machine to another. This was not considered in this work and should be part of future work. Another desirable feature missing in both approaches is the enforcement of a user quality of service requirement such as budget and deadline. For instance, a user may want to minimize the execution cost and still require that the execution finishes before a predefined deadline.

The leasing model considered in the cloud algorithm is a simplified version compared to the current pricing policies of cloud providers. For instance, providers define different zones in which virtual machines can be started and the price varies from zone to zone. Additionally, the cost of transferring data from one instance to the other as well as the storage services offered should be considered when estimating the total cost.

Finally, a workflow management system does not necessarily have to select resources from a single source. Instead, it can have access to resources in two different ways; the first one is through a static pool of resources and the second one through a public cloud provider. The static pool would be composed of private resources or any other resource that is available for use (for example an instance previously leased from an IaaS provider). This set of resources meets the requirements of the algorithm for grid environments. On the other hand, the resources offered by the public cloud provider meet the requirements of the algorithm for cloud environments. This implies that both algorithms could be integrated in such way that the utilization of the resources is optimal in terms of the scheduling objective (cost minimization, time minimization, etc.). In such scenario, the static resources would be used to schedule the workflow tasks, and if these are unable to meet the QoS requirement or scheduling objective then the algorithm would turn to the public cloud resources to successfully finish the execution of the application.

References

[1] A. Lazinica. *Particle Swarm Optimization*, Ed. In- Tech. Vienna (Austria). 2009. pp. 1-486.

[2] J. Kennedy and R. C. Eberhart, ”Particle swarm optimization”, Proceedings of the 1995 IEEE International Conference on Neural Networks, vol. 4, 1942–1948. IEEE Press.

[3] Suraj Pandey, Linlin Wu, Siddeswara Mayura Guru, Rajkumar Buyya, "A Particle Swarm Optimization-Based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments," aina, pp.400-407, 2010 24th IEEE International Conference on Advanced Information Networking and Applications, 2010.

[4] H. Yoshida, K. Kawata, Y. Fukuyama, and Y. Nakanishi. A particle swarm optimization for reactive power and voltage control considering voltage stability. In *the International Conference on Intelligent System Application to Power System*, pages 117–121, 1999.

[5] C. u. O. Ourique, E. C. J. Biscaia, and J. C. Pinto. The use of particle swarm optimization for dynamical analysis in chemical processes. *Computers and Chemical Engineering*, 26(12):1783–1793, 2002.

[6] T. Sousa, A. Silva, and A. Neves. Particle swarm based data mining algorithms for classification tasks. *Parallel Computing*, 30(5-6):767–783, 2004.

[7] J. D. Ullman. Np-complete scheduling problems. *J. Comput.Syst. Sci.*, 10(3), 1975.

[8] T. A. Feo and M. G. C. Resende, Greedy Randomized Adaptive Search Procedures, *Journal of Global Optimization*, 6:109-133, 1995.

[9] S. Binato et al., A GRASP for job shop scheduling. *Essays and surveys on meta-heuristics*, pp.59-79, Kluwer Academic Publishers, 2001.

[10] J. Blythe et al., Task Scheduling Strategies for Workflow-based Applications in Grids, *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, 2005

[11] J. Blythe and et al., Task scheduling strategies for workflow based applications in grids. In Proc. of CCGRid ’05.

[12] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[13] N. Metropolis et al., Equations of state calculations by fast computing machines. *Journal of Chemistry and Physics,* 21:1087-1091, 1953.

[14] L. Young et al., Scheduling Architecture and Algorithms within the ICENI Grid Middleware, *UK e-Science All Hands Meeting*, IOP Publishing Ltd, Bristol, UK, Nottingham, UK, Sep. 2003, pp. 5-12.

[15] P. J. M. Van Laarhoven, E. H. L. Aarts, and J. K. Lenstra. Job shop scheduling by simulated annealing. Operations Research, 40:113{125, 1992.

[16] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[17] H. H. Hoos and T. StÄutzle, *Stochastic Local Search: Foundation and Applications*, Elsevier Science and Technology, 2004.

[18] A. S. Wu, et al., An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling, *IEEE Transactions on Parallel and Distributed Systems*, 15(9):824- 834, September 2004.

[19] A. Y. Zomaya, C.Ward, and B. Macey, Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues, *IEEE Transactions on Parallel and Distributed Systems*, 10(8):795-812, Aug. 1999.

[20] L. Wang et al., Task Mapping and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach, *Journal of Parallel and Distributed Computing,* 47:8-22, 1997.

[21] Rubing Duan, Thomas Fahringer, Radu Prodan, Jun Qin, Alex Villazon, and Marek Wieczorek. Real World Workflow Applications in the Askalon Grid Environment. In European Grid Conference (EGC 2005), Lecture Notes in Computer Science. Springer Verlag, February 2005

[22] M. Wieczorek, R. Prodan, and T. Fahringer, Scheduling of Scientific Workflows in the ASKALON Grid Environment, *ACM SIGMOD Record*, 34(3):56-62, Sept. 2005.

[23] A. Salman. Particle swarm optimization for task assignment problem. *Microprocessors and Microsystems*, 26(8):363– 371, November 2002.

[24] L. Zhang, Y. Chen, R. Sun, S. Jing, and B. Yang. A task scheduling algorithm based on pso for grid computing. *International Journal of Computational Intelligence Research*, 4(1), 2008.

[25] The Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne. Cloudbus Workflow Engine. http://cloudbus.org/workflow.

[26] The Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne. http://cloudbus.org.

[27] JSwarm-PSO. http://jswarm-pso.sourceforge.net.

[28] L. Kleinrock, "An Internet Vision: The Invisible Global Infrastructure," *Ad Hoc Networks*, vol. 1, no. 1, pp. 3–11, July 2003.

[29] Buyya, R. and Venugopal, S.(2009) Market-Oriented Computing and Global Grids: An Introduction, in Market-Oriented Grid and Utility Computing (eds R. Buyya and K. Bubendorfer), John Wiley & Sons, Inc., Hoboken, NJ, USA.

[30] Foster, I., Kesselman, C., Nick, J. M. and Tuecke, S. (2003) The Physiology of the Grid, in Grid Computing: Making the Global Infrastructure a Reality (eds F. Berman, G. Fox and T. Hey), John Wiley & Sons, Ltd, Chichester, UK. doi: 10.1002/0470867167.ch8

[31] The Globus Alliance. http://www.globus.org/. Accessed October 2011.

[32] M. Baker, R. Buyya, and D. Laforenza, Grids and Grid technologies for wide-area distributed computing, Software: Practice and Experience 32(15):1437–1466 (Dec. 2002).

[33] I. Foster, C. Kesselman, and S. Tuecke, The anatomy of the Grid: Enabling scalable virtual organizations, International Journal of High Performance Computing Applications, 15(3):200–222 (2001).

[34] R. Buyya, D. Abramson, and J. Giddy, An economy driven resource management architecture for global computational power Grids, Proc. 7th International Conf. Parallel and Distributed Processing Techniques and Applications, Las Vegas, June 26–29, 2000.

[35] S. Pandey, Scheduling and Management of Data Intensive Application Workflows in Grid and Cloud Computing Environments. 2010.

[36] Buyya, R., Broberg, J., and Goscinski, A. (ed), Cloud Computing Principles and Paradigms, Wiley, NJ, USA, 2011.

[37] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya, Aneka: A Software Platform for .NET-based Cloud Computing, High Speed and Large Scale Scientific Computing, 267-295pp, W. Gentzsch, L. Grandinetti, G. Joubert (Eds.), ISBN: 978-1-60750-073-5, IOS Press, Amsterdam, Netherlands, 2009.

[38] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, Cloud Computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems, 2009.

[39] SalesForce. http://www.salesforce.com/au/saas/. May 2011.

[40] P. Mell and T. Grance, The NIST Definition of Cloud Computing, National Institute of Standards and Technology, Information Technology Laboratory, Technical Report Version 15, 2009.

[41] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. SIGMOD Record, 34(3), 2005.

[42] V. Hamscher et al. Evaluation of Job-Scheduling Strategies for Grid Computing. In 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000), Springer-Verlag, Heidelberg, Germany, 2000; 191-202.

[43] S. Pandey, W. Voorsluys, M. Rahman, R. Buyya, J. Dobson, and K. Chiu, A Grid Workflow Environment for Brain Imaging Analysis on Distributed Systems, in *Concurrency and Computation: Practice and Experience*, 21(16):2118-2139, Wiley Press, New York, USA, November 2009.

[44] J. Yu and R. Buyya. A Novel Architecture for Realizing Grid Workflow using Tuple Spaces. In 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004), Pittsburgh, USA, IEEE CS Press, Los Alamitos, CA, USA, Nov. 8, 2004.

[45] S. Venugopal, K. Nadiminti, H. Gibbins, and R. Buyya, "Designing a Resource Broker for Heterogeneous Grids," *Software: Practice and Experience,* vol. 38, pp. 793-825, July 10 2008.

[46] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing," *Proceedings of Cloudcomp 2009*, Munich, Germany: 2009.