TASK MANAGEMENT DRIVEN SIMULATION OF GRID APPLICATIONS USING SIMGRID

by

Uday Kumar B. Narayanappa, B.E.

A Thesis

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCES

Approved

Dr. Noe Lopez Benitez
Chair of Committee

Dr. Yu Zhuang

Dr. Sunho Lim

Accepted

Ralph Ferguson
Dean of the Graduate School

December, 2010

Acknowledgements

I would like to express my sincere thanks for the help, guidance and support received from my thesis advisor, Dr. Noe Lopez-Benitez. He was always available and willing to help me finish this thesis. Special thanks and sincere gratitude are due for his accepting me as his thesis student and helping me complete this endeavour.

Appreciation is extended to Dr. Yu Zhuang and Dr. Sunho Lim for being part of my thesis committee. I would like to thank fellow graduate student Santhosh Aditham for sharing his knowledge and for his disposition to help me during the development of this work. I would like to also thank SimGrid for clearing my doubts on the SimGrid simulation tool. The SimGrid forum and the SimGrid website helped in gathering the information for my research.

Table of Contents

Abstract

Task graph models are normally used to represent the organization of grid applications in terms of the interaction between tasks. However, in a grid environment there is neither an efficient task management scheme nor any efficient structures to support the execution of large complex applications. This thesis proposes a simulation model that integrates a task management scheme and the libraries provided by the SimGrid simulation tool to evaluate the performance of a complex application submitted for execution to a grid-computing environment under a given scheduling heuristic scheme. The efficient structures are used by the task management scheme to monitor the state of the execution of completed tasks. The task management scheme uses an alternative representation to the task graph model referred to as the functional description of the application. The proposed simulation model takes the functional description of a complex application as input and outputs the total execution time of the complex application under a given scheduling heuristic scheme. Different task scheduling heuristic schemes are analyzed in this thesis, using SimGrid as the simulation engine. The results reported compare the performance of a complex application under different scheduling heuristic schemes.

List of Tables

List of Figures

Chapter 1

Introduction

Complex computation jobs can be broken down into several modules called tasks. Task graph models are normally used to represent the organization of grid applications in terms of the interaction between tasks. However, in a grid environment there is neither an efficient task management scheme nor any efficient structures to support the execution of large complex applications. So, there is a need for efficient structures that can be used as an alternative to task graph models and there is also a need for a task management scheme to support the execution of large complex applications submitted to a grid-computing environment under a given scheduling heuristic scheme. The efficient structures are used by the task management scheme to monitor the state of execution of completed tasks.

This thesis proposes a simulation model that integrates a task management scheme and the libraries provided by the SimGrid simulation tool to evaluate the performance of a complex application submitted for execution to a grid-computing environment under a given scheduling heuristic scheme.

The simulation model is implemented in the C programming language. Different task scheduling heuristic schemes are analyzed in this thesis, using SimGrid as the simulation engine. SimGrid helps in achieving accurate and validated simulation results.

The task management scheme uses an alternative representation to the task graph model referred to as the functional description of the application. The functional description is a structure that captures the topology of the task graph; it consists of a set of queue structures, where each queue represents a path in the task graph. The proposed simulation model uses the functional description as it can be specified from the task graph model of the application, or it can be used to reflect the code organization of an application, particularly where functional programming languages are used.

Simulation helps in studying the complex interaction of a system and finding out how certain changes in the environment can affect the operation of the system. It also helps in testing new features before they are implemented in the actual system. The proposed simulation model takes the functional description of a complex application as input and outputs the total execution time of the complex application under a given scheduling heuristic scheme. The results reported compare the performance of a complex application under different scheduling heuristic schemes. The simulation process assumes a distribution of execution times as selected by the user.

This thesis is organized as follows. In Chapter 2, the basic task graphs, functional description of the task graphs and Monte Carlo simulation are presented. Chapter 3 presents the overall idea of the task management. Chapter 4 discusses the history of SimGrid, components of SimGrid, fundamental concepts of SimGrid, how to build the

SimGrid simulation and SimGrid functions used in the thesis. Chapter 5 discusses the overview of the simulation model, data used in the simulation model and the detailed explanation of modules in the simulation model. Chapter 6 introduces several task scheduling heuristic schemes which are compared in this thesis. Chapter 7 details the implementations of the proposed simulation model. Chapter 8 discusses the experimentation procedure, results and analysis. Chapter 9 details the conclusions and future work. The appendix provides the code for the implementation.

Chapter 2

Background Information

This chapter discusses in detail the background information of the task graph, the functional description of the task graph and the Monte Carlo simulation.

2.1 Task Graph

A task graph is a Directed Acyclic Graph (DAG) where vertices of the graph represent tasks and the directed edges between vertices of the graph represent the dependencies between tasks. Tasks may be of variable size and may have multiple input dependencies resulting in varying execution times. Tasks create output that forms the input to many other (dependent) tasks [5]. Complex computation jobs can be broken down into several modules called tasks and tasks are executed according to some precedence constraints [2].

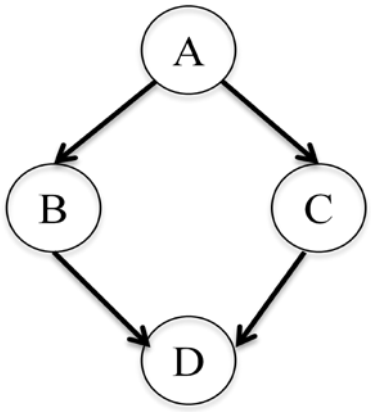

Fig. 2.1 A task graph with dependencies

Fig. 2.1 shows a task graph with dependencies. Tasks B and C depend on the task A and the task D depends on tasks B and C. If the task A completes the execution, then tasks B and C will start the execution at the same time. Tasks B and C should complete the execution for the task D to start the execution as it depends on them. Once the task D completes the execution, the task graph in Fig. 2.1 has completed execution.

The task graph can be a series-parallel or a non-series-parallel task graph. The following examples identify a series-parallel task graph [3, 4, 6].

1. A single edge is a series-parallel graph.
2. If G' and G'' are both series parallel graphs, then Fig. 2.2 shows the series combination of G' and G'' is a series-parallel graph.
3. If G' and G'' are both series parallel graphs, then Fig. 2.3 shows the parallel combination of G' and G'' is a series parallel graph.

The task graph shown in Fig. 2.4 is an example of a series-parallel task graph and the task graph shown in Fig. 2.5 is an example of a non series-parallel task graph [3]. The sub graphs ABDF and ACEF in Fig. 2.4 are in series combination and they are in parallel combination with each other. The sub graph ABDF in Fig. 2.5 is not in parallel with the sub graph ACEF, as there is a dependency between tasks B and E [3].

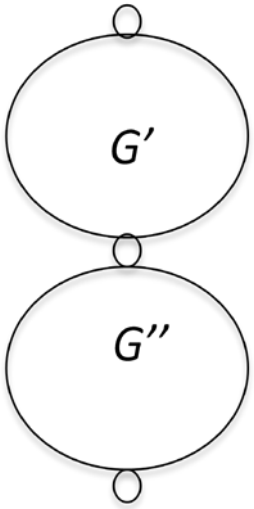

Fig. 2.2 Series combination of G' and G''

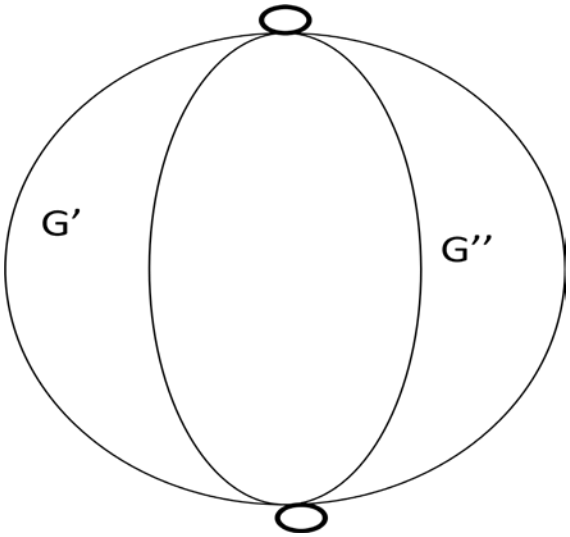

Fig. 2.3 Parallel combination of G' and G''

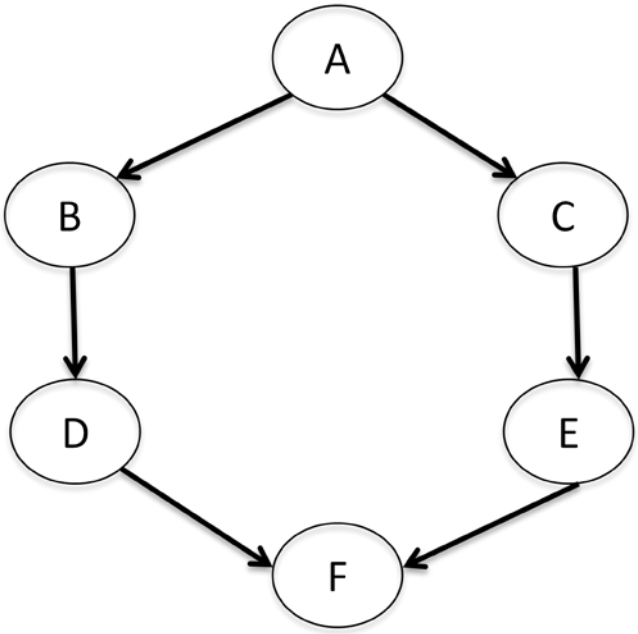

Fig. 2.4 Series-parallel task graph

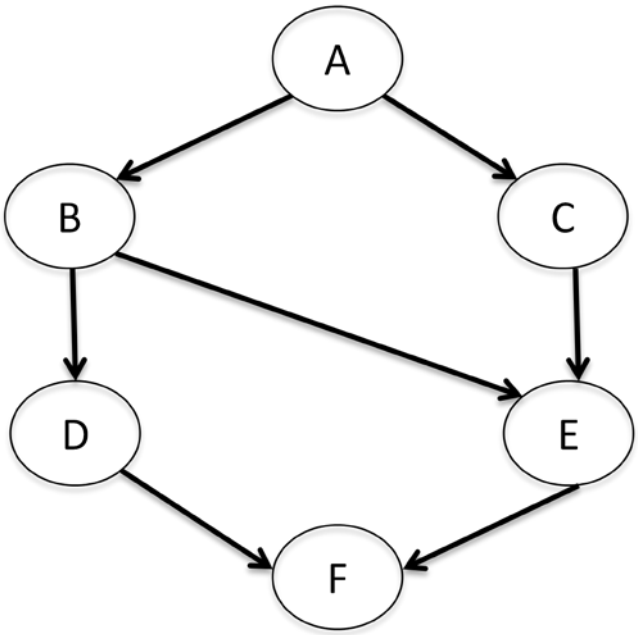

Fig. 2.5 Non series-parallel task graph

2.2 Functional Description of Task Graphs

Task graph models are normally used to represent the organization of grid applications in terms of the interaction between tasks. The task graph helps in identifying tasks that can be executed in parallel and also helps in identifying sequential task execution paths [1].

The functional description is a structure that captures the topology of the task graph; it consists of a set of queue structures, where each queue represents a path in the task graph.

The mapping scheme presented in paper [1] assumes that any task is regarded as a function applied to data produced by all its predecessor tasks after execution. For scheduling purposes, a task graph model can be mapped into a functional description resulting in simpler specification of a complex application. Consider T, which represents the task graph model of a given complex application and it is partitioned into M tasks expressed with the following M-tuple:

$$T = [T_1 (), T_2 (), T_3 (), \ldots , T_{F1} (), T_{F2} (), \ldots , T_{Fn} ()] \quad (1)$$

Where,

T_i () identifies tasks that are not terminal nodes in T

T_{Fj} (), $1 \leq j \leq n$ identifies n terminal nodes in T

Tasks are considered as functions that depend on the execution of its predecessor tasks and works on the data provided within parenthesis. Square brackets are used to represent the dependencies on the execution of previous tasks.

Two steps in the mapping process as reported in paper [1] are as follows: -

1. A sequence function is used to represent each node in the task graph.
2. Group all terminal nodes into a functional description as follows:

$$T = [T_{F1} , T_{F2} , \ldots , T_{Fn}] \quad (2)$$

where each terminal node is also expressed as a sequence function with all its predecessor sequence functions as arguments i.e., $T_{Fi} [T_a [T_b [. . .]. . .]]$, where T_a and T_b are tasks that lie on the same execution path in the task graph that leads to T_{Fi}. Each sequence function represents a possible execution path with at least one initial task. The construct in equation (2) is an alternative model to the task graph T. Each path in equation (2) leads to a root that corresponds to an initial task or a task that has no incoming arcs. A set of all root tasks is selected for execution in parallel [1].

Consider a task graph T as shown in Fig. 2.6 [1]. Task graph T has 7 tasks labeled A, B, C, D, E, F and G and the arcs show the dependencies between tasks. After applying the above mapping scheme to a task graph T, a second representation of T is obtained and it is shown in Fig. 2.7 [1] and the following functional description is generated:

$$T[F[D[B[A]]], G[D[B[A]], E[B[A]], C[A]]] \quad (3)$$

Tasks F and G can be executed in parallel as long as the preceding tasks have been executed. From equation (3), the task A is the root node in each path and it is executed. On completion of the task A, the equation in (3) is reformulated into new one as shown below:

T[F[D[B]], G[D[B], E[B], C]]

This shows that tasks B and C are root tasks. Tasks B and C can be executed in parallel and on completion of tasks B and C; the functional description is changed into new construct as shown below:

T[F[D], G[D, E]]

The above construct shows that tasks D and E can be executed in parallel and on completion leads to the final construct:

T[F, G]

This construct shows that tasks F and G can be executed in parallel, which are the terminal nodes in equation (3). The functional description can be specified from the task graph model of the application, or it can be used to reflect the code organization of an application, particularly where functional programming languages are used.

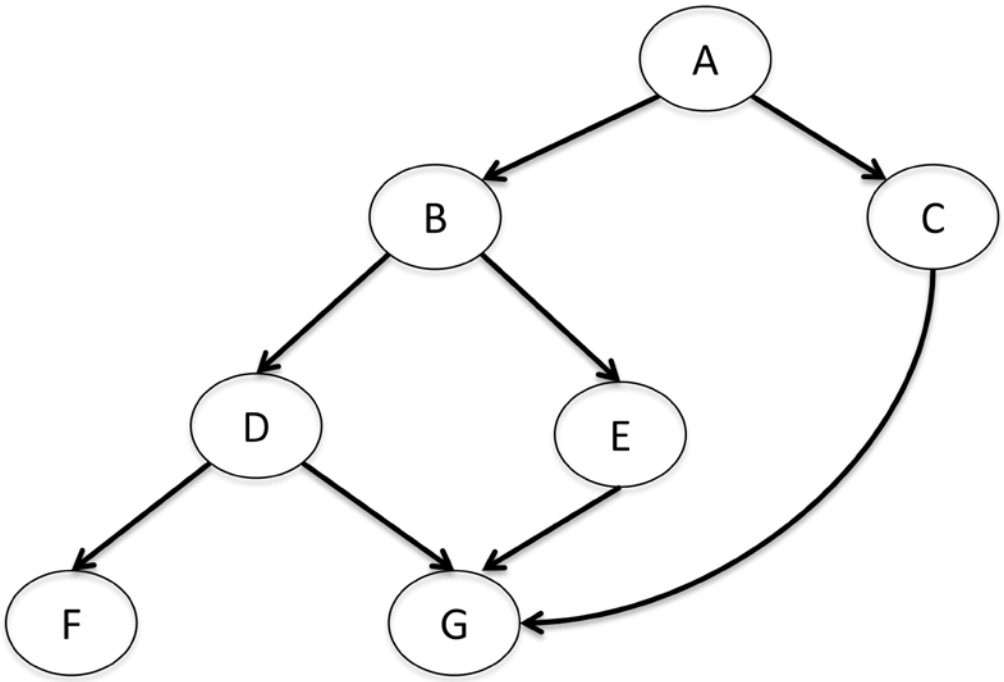

Fig. 2.6 Task graph T with dependencies

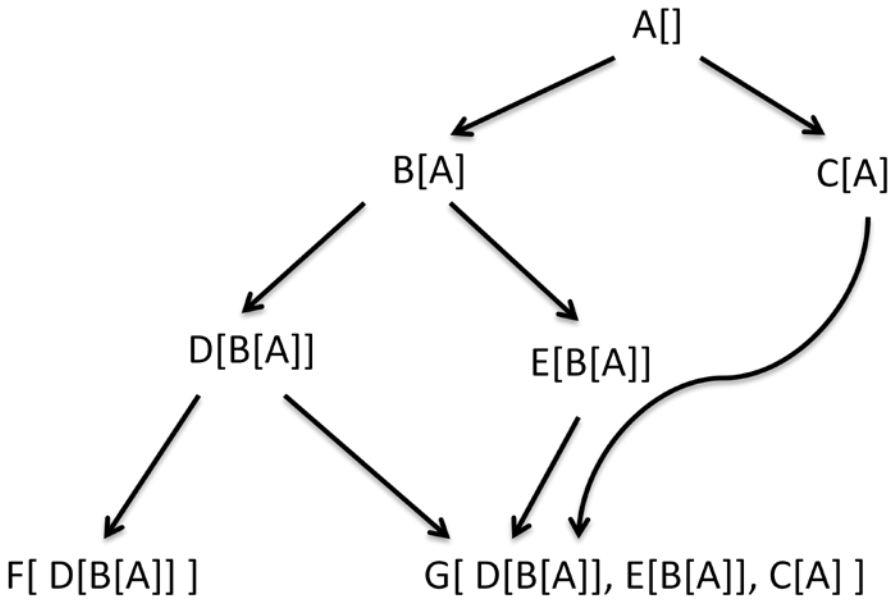

Fig. 2.7 Mapping tasks into the functional description

2.3 Monte Carlo Simulation

Mote Carlo (MC) simulation is called stochastic simulation - meaning it is based on the use of probability statistics and random numbers to perform simulation [7, 8, 9]. In MC method, the system can be simulated directly without any need for differential equations that describe the behavior of the system. MC method needs only the description of the system with probability density functions (pdfs) [3].

In this thesis, evaluating a complex application represented by functional description involves estimating the total execution time of the complex application given the execution times of its tasks. The execution times of tasks are random variates with specified pdfs. MC method is used to analyze a task graph by generating random or pseudo-random numbers [3]. The random numbers in MC method mean independent random numbers distributed uniformly in the interval (0, 1). Several algorithms are proposed for generating random variates for many distributions. This thesis provides three different procedures for generating the random variates for three kinds of distribution: exponential, normal and uniform distribution.

2.3.1 Generation of random variates from an exponential distribution

The summarized algorithm for generating a random variate from an exponential distribution is as follows [3, 6, 7, 8]:

1. Generate U from uniform distribution ranged (0, 1).
2. $X = - \beta \times \ln U$ where β is mean and $\beta > 0$.

3. Output X.

2.3.2 Generation of random variates from a normal distribution

The summarized algorithm for generating a random variate from a normal distribution is as follows [3, 6, 7, 8, 10]:

1. Generate 12 uniformly distributed random variates U_1, U_2 , U_3 , …, U_{12} ranged (0, 1).
2. $Z = \sum_{i=1}^{12} U_i - 6.$
3. $X = \mu + \sigma Z$ where μ is mean and σ is standard deviation.
4. Output X.

2.3.3 Generation of random variates from a uniform distribution

The summarized algorithm for generating a random variate from a uniform distribution is as follows [3, 6, 7, 8]:

1. Generate U from uniform distribution ranged (0, 1).
2. $X = U \times (e - s) + s$ where s is the starting point and e is the ending point.
3. Output X.

The Total Execution Time (TET) of the entire set of tasks in the functional description is estimated in this thesis to evaluate the performance of a complex application. The simulation is repeated for a number of cycles. The total execution time is estimated from each simulation cycle directly from the generated random variates.

Finally, the Mean of the Total Execution Time (MTET) is calculated within a 99% confidence interval.

Chapter 3

Task Management

This chapter presents the overall idea of the task management scheme. The task management is a critical component for computational grids. Tasks are assigned to nodes in the grid environment based on resources needed for their computation and available resources on nodes. A grid has to manage lot of resources as the storage spaces and the computational power [14].

Mapping the task graph into functional description can generate execution sequences of tasks for the general purpose-computing network. The network signals the end of execution of a task currently in execution. Execution profile is a structure that captures the current execution state of tasks in the grid environment [1]. Initially, the execution profile is the same as the functional description of a complex application. The execution profile changes as tasks are executed.

In a task graph model, the task that has no incoming arcs is called a root task. Let R be the dynamic set of root tasks in the execution profile. The set R contains tasks that are not yet scheduled for execution and/or tasks in execution. A dedicated server node or the submitting node can be used as a local Task Manager (TM). The working of TM as presented in paper [1] is as follows:

1. Initially, TM uses the functional description of a complex application as input and creates an execution profile that is the same as the functional description.
2. TM identifies the set of root tasks R in the current execution profile.
3. TM notifies the set R to the task scheduler referred to as Meta Scheduler (MS) to schedule tasks in set R, which are not yet scheduled.
4. Once tasks are scheduled, TM waits for execution completion notification of a task from MS.
5. TM updates the current execution profile and the set R by removing the task that just completed execution.
6. The process of identifying the set of root tasks R, notifying the set R to MS, waiting for task execution completion notification, and updating the execution profile (2 - 5) is repeated as long as the set R is not empty.

Meta Scheduler schedules tasks in the grid environment. MS has information about the network topology. In this thesis, a meta scheduler file is used that has information about execution times of tasks, communication costs between nodes in the network, and the data dependency between tasks. A local level or the resource scheduling service provider can be used as MS. The working of MS as presented in paper [1] is as follows:

1. Receives the notification from TM about the set of root tasks R, which are yet to be scheduled.

2. MS schedules tasks in set R which are not yet scheduled on some node in the network using any task scheduling heuristic scheme.
3. MS waits for task execution completion notification from the node in the network.
4. MS interacts with TM and notifies the task execution completion.

Thus, the Task Manager and the Meta Scheduler do the necessary task management. The overall task management scheme used for scheduling and executing tasks in the grid environment can be summarized as follows [1]:

While $R \neq \varnothing$

{

Submit all $T_i \in R$ not yet scheduled

Flag each T_i as scheduled

Check for task execution completion notification events

Update execution profile

Update R:

Remove tasks executed

Add new root tasks

}

Consider the task graph T that represents a complex application as shown in Fig. 2.6. Equation (3) shows the functional description of the complex application. The initial execution profile of T is the same as the functional description as shown:

T[F[D[B[A]]], G[D[B[A]], E[B[A]], C[A]]]

The initial set R consists of only A. i.e. R = {A}. Let S_i be a flag to indicate the task T_i has been scheduled.

Submit A

$S_A = 1$

Check for task execution completion notification events

If A completes execution then:

Update the execution profile:

T[F[D[B]], G[D[B], E[B], C]]

Update the root tasks set: R = {B, C}

The local TM uses the updated execution profile and the new set of root tasks R and issues the following actions:

Submit B and C

$S_B = 1$; $S_C = 1$

Check for task execution completion notification events

At least one execution completion event causes the new iteration:

If B has been executed then:

Update the execution profile:

T[F[D], G[D, E, C]]

Update the root tasks set: R = {C,D,E}

If C has been executed then:

Update the execution profile:

T[F[D[B]], G[D[B], E[B]]]

Update the root tasks set: R = {B}

Assume that task C completes first and then task B completes. Then the new execution profile is T[F[D], G[D, E]] and the root tasks set R = {D, E}. The new iteration schedules tasks D and E. If the task E completes first and then the task D completes, then the resulting execution profile is T[F, G] and R = {F, G}. Tasks F and G are terminal nodes in the task graph and scheduling these tasks will make the set R empty i.e. R = {∅}. This completes the execution of the complex application represented by the functional description in equation (3).

The task management scheme uses the execution profile created from the functional description of a complex application. It is possible to reach the maximum level of parallelism as tasks are executed as soon as its last predecessor has been executed.

Chapter 4

SimGrid as a Simulation Tool

Many software tools have been developed to build and run simulations in the grid environment domain. Several software libraries and the environment support the generic discrete-event simulations [15, 16, 17]. Tools are available for simulating the parallel applications on distributed systems [18]. Microgrid [19] is one of the available tools that are used for simulating grid applications. Table 4.1 shows the comparison of existing tools for experimental large-scale distributed computing research [20]. SimGrid is a toolkit that provides core functionalities for the simulation of distributed applications in heterogeneous distributed environments [21]. SimGrid is a generic framework for large-scale distributed experiments [20]. SimGrid helps in achieving accurate and validated simulation results.

4.1 History of SimGrid

The first version of SimGrid as reported in paper [18] was a discrete-event simulation toolkit. SimGrid's functionalities can be used to build simulators for particular computing environment topologies and/or particular application domains. But the number of functionalities provided in first version is limited.

Table 4.1 Comparison of existing tools

	CPU	**Disk**	**Network**	**Application**	**Requirement**	**Settings**	**Scale**
Grid'5000 [4]	direct	direct	direct	direct	access	fixed	< 5000
PlanetLab [8]	virtualize	virtualize	virtualize	virtualize	none	uncontrolled	< 850
ModelNet [20]	-	-	emulation	emulation	lot material	controlled	100 nodes/real host
MicroGrid [21]	emulation	-	fine d.e.s.	emulation	none	controlled	few 100
ns2 [1]	-	-	fine d.e.s.	coarse d.e.s.	C++ and Tcl	controlled	<1,000 [18]
SSFNet [9]	-	-	fine d.e.s.	coarse d.e.s.	Java	controlled	<100,000 [18]
GTNetS [18]	-	-	fine d.e.s.	coarse d.e.s.	C++	controlled	<177,000 [18]
ChicSim [17]	coarse d.e.s.	-	fine d.e.s.	coarse d.e.s.	C	controlled	thousands
OptorSim [2]	coarse d.e.s.	amount	math/d.e.s.	coarse d.e.s.	Java	controlled	few 100
GridSim [5]	coarse d.e.s.	fine d.e.s.	fine d.e.s.	coarse d.e.s.	Java	controlled	few 100
PlanetSim [11]	-	-	constant time	coarse d.e.s.	Java	controlled	100,000 [15]
PeerSim [12]	-	-	-	state machine	Java	controlled	1,000,000 [15]
SimGrid	coarse d.e.s.	-	math/d.e.s.	d.e.s./emul	C or Java	controlled	few 10 000

In second version new software layer is added to provide high-level abstractions and it provides two interfaces:

1. SG: A basic low-level toolkit.
2. MSG: More application-oriented simulator built on SG.

The paper [20] introduces us to the third version of SimGrid in which many features were added with respect to previous versions. It supports dynamic resource availabilities and failures. The new simulation engine has better speed, modularity and scalability.

4.2 Overview of SimGrid Components

Fig. 4.1 shows the current SimGrid software stack with its relevant components.

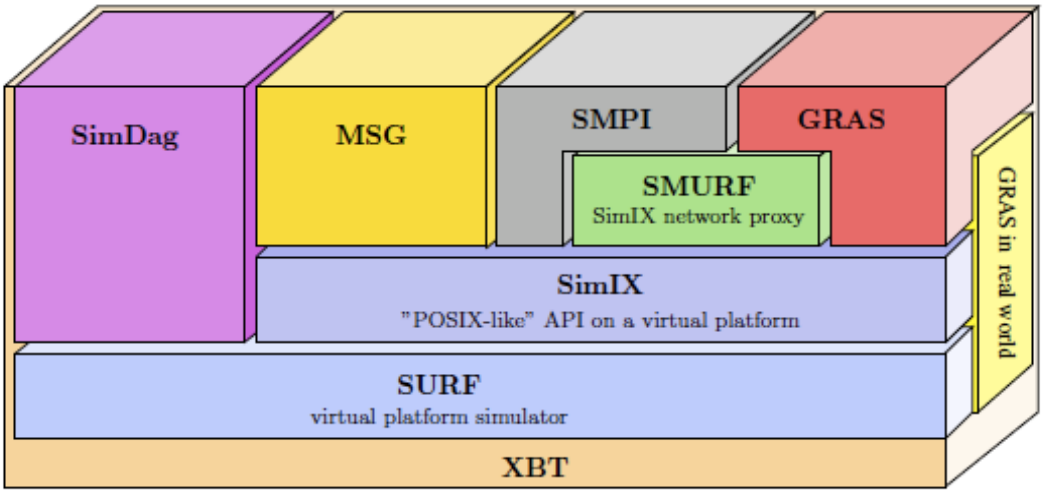

Fig. 4.1 Overview of SimGrid components

SimGrid offers four interfaces:

1. SimDag: Used to investigate scheduling heuristics for the task graph applications.
2. MSG: Used to study the concurrent sequential processes.
3. SMPI: Used to directly simulate the MPI applications.
4. GRAS: Uses SimGrid as a development lab for the real distributed applications.

The details of interfaces as reported in paper [20] are as follows: SimDag and MSG are interfaces used for research purposes. GRAS and SMPI are interfaces used for development purposes. A toolbox module called XBT implements data containers, logging and exceptional mechanisms and support for configuration and portability. SURF is the simulation engine and it is the SimGrid simulation kernel. Internal module SimIX provides Application Programming Interface (API) on top of SURF. SMURF module is used to harness the memory of several computers, which improves the scalability of SimGrid.

Currently, SimGrid is very active project in terms of research and development. It is freely available on website with useful example programs and tutorials. It can be ported on Linux, Windows, Mac OS X and AIX [21].

4.3 Fundamental Concepts of SimGrid

Following are the core abstractions implemented in SimGrid as presented in paper [18]:

1. Agent: Agent is an entity responsible for making scheduling decisions. A code, private data and the location at which it executes constitute an agent.
2. Location or Host: A place in the network topology at which an agent runs. Computational resource and mailboxes defines the location that enables communication with other agents.
3. Task: A task is defined by some computing amount, private data and data size. It can be a computation or a data transfer.
4. Path: A path is a collection of communication resources that represents a set of physical network links. Locations or Hosts are interconnected through paths.
5. Channel: Channel helps in communication between agents. A channel contains information about communication ports used by agents at locations.

With the help of these abstractions the simulation algorithms in SimGrid should be defined interns of agents running at locations and the interaction between agents takes place by sending, receiving and executing the simulated application tasks.

4.4 Building a SimGrid Simulation

In this thesis, one of the SimGrid modules - SimDag is used for building the simulation tool. Steps followed in SimGrid programs are [18]:

1. Modeling the application: Defines the code of each agent. Several functions exist in SimGrid library for modeling agents such as SD_task_create, SD_task_get_data, SD_task_set_data.
2. Modeling of the physical platform: It consists in creation of resources and definitions of links, hosts and routing table that specify paths. Reading a platform description file can do it or by using a function SD_create_environment.
3. Deployment of application: Agents are created and allocated to hosts.
4. Simulation: Simulation can be launched using the function SD_simulate.

4.5 SimGrid Functions Used

Several functions are available in SimGrid for managing workstations (nodes in the network), managing the network links, managing tasks, managing task dependencies, creating the environment and launching the simulation. Detailed descriptions of functions used in this thesis are as follows [21]:

4.5.1 SD_workstation_get_list

Returns the workstation list.

const SD_workstation_t * SD_workstation_get_list(void)

4.5.2 SD_workstation_set_access_mode

Set the access mode for workstations that execute tasks. By default, the access mode is shared, i.e. several tasks can be executed at the same time on the workstation. In this case, power of the workstation is shared among running tasks.

void SD_workstation_set_access_mode(SD_workstation_t workstation, e_SD_workstation_access_mode_t access_mode)

where workstation – a node in the grid network

access_mode - SD_WORKSTATION_SHARED_ACCESS or SD_WORKSTATION_SEQUENTIAL_ACCESS .

4.5.3 SD_task_create

Creates a new task.

SD_task_t SD_task_create (const char * name, void* data, double amount)

where name – name of the task

data – user data associated with the task

amount – amount of the task

4.5.4 SD_task_get_name

Returns the name of the task.

const char* SD_task_get_name (SD_task_t task)

where task – a task

4.5.5 SD_task_schedule

Schedules a task

void SD_task_schedule (SD_task_t task, int workstation_count, const SD_workstation_t * workstation_list, const double * computation_amount, const double * communication_amount, double rate)

where task – a task to schedule

workstation_count – number of workstations on which the task will be executed

workstation_list – workstations on which the task will be executed

computation_amount – computation amount for each workstation

communication_amount – communication amount between each pair of workstations

rate – task execution speed rate

4.5.6 SD_task_get_state

Returns the state of the task. The current state can be SD_NOT_SCHEDULED, SD_SCHEDULED, SD_RUNNABLE, SD_RUNNING, SD_DONE OR SD_FAILED.

e_SD_task_state_t SD_task_get_state (SD_task_t task)

where task – a task

4.5.7 SD_task_get_start_time

Returns the start time of the task.

double SD_task_get_start_time (SD_task_t task)

where task – a task

4.5.8 SD_task_get_finish_time

Returns the finish time of the task.

double SD_task_get_finish_time (SD_task_t task)

where task – a task

4.5.9 SD_task_dependency_add

Adds a dependency between two tasks.

void SD_task_dependency_add (const char * name, void* data, SD_task_t src, SD_task_t dst)

where name – the name of new dependency

data – user data with the dependency

src – the task that must be executed first

dst – the task that depends on src

4.5.10 SD_task_watch

Adds a watch point to the task.

void SD_task_watch (SD_task_t task, e_SD_task_state_t state)

where task – a task

state – the state to watch. State can be SD_NOT_SCHEDULED, SD_SCHEDULED, SD_RUNNABLE, SD_RUNNING, SD_DONE OR SD_FAILED.

4.5.11 SD_init

This function must be called before calling any other function and it initializes SD internal data.

void SD_init (int * argc, char ** argv)

where argc – argument number

argv - argument list

4.5.12 SD_create_environment

Creates an environment.

void SD_create_environment (const char * platform_file)

where platform file – a XML file that has data stored in it to create an environment.

4.5.13 SD_get _clock

Returns the current clock time in seconds.

double SD_get_clock (void)

4.5.14 SD_simulate

Launches the simulation

xbt_dynar_t SD_simulate (double how_long)

where how_long – maximum duration of the simulation (negative value means no limit imposed on time)

4.5.15 SD_exit

Deletes all SD internal data. This function is called when the simulation is finished.

void SD_exit (void)

Chapter 5

Simulation Model

Naylor [7] defined simulation as follows: “Simulation is a numerical technique for conducting experiments on a digital computer, which involves certain types of mathematical and logical models that describe the behavior of systems or games over extended periods of real time” [p. 6].

A system is a set of related elements. These elements will have relationships that can be either internal or external. Internal relationships combine elements within the system and the external relationships combine elements with the environment. In order to study a system, it is necessary to build a scientific model. Simulation is a numerical method of finding a solution to the model. The numerical method approximates solution by substituting numerical values for variables and parameters [7].

Simulation helps in studying the complex interaction of a system and to find out how certain changes in the environment can affect the operation of the system. It also helps in testing new features before they are implemented in the actual system. Simulation performed in computers has an advantage that experiments can be repeated without much risk [3].

5.1 Data used in Simulation Model

The following notation is used to describe the simulation model and related issues:

- The number ***k*** of subtasks in a task graph.
- A task graph G (T, E) where the vertex set $T = \{T_1, T_2, \ldots, T_k\}$ consists of k tasks which compose some complex application and the edge set E consists of ordered pairs from T which correspond to data or control.
- The number ***n*** of nodes composing a grid environment.
- Set of nodes $W = \{W_1, W_2, \ldots, W_n\}$.
- A k x k task data dependency matrix $Pkt[i, j]$, $1 \leq i, j \leq k$ where pkt_{ij} is the average number of data packets of standard size that is sent from T_i to T_j. Alternatively, these can be specified as edge weights for the elements of E.
- A k x n execution time matrix $B[i, j]$, $1 \leq i \leq k$, $1 \leq j \leq n$ where b_{ij} is the average execution time of T_i on W_j.
- A n x n communication cost matrix $C[r, s]$, $1 \leq r, s \leq n$ where each entry c_{rs} is the average communication cost to transfer a data packet of standard size from W_r to W_s.
- A k x n task allocation matrix $A[i, j]$, $1 \leq i \leq k$, $1 \leq j \leq n$ where entry $a_{ij} = 1$ if T_i has been allocated to W_j, and 0 otherwise.
- S_i, $1 \leq i \leq k$, be a flag to indicate the task T_i has been scheduled.
- C_i, $1 \leq i \leq k$, be a flag to indicate the task T_i has completed execution.

- Set of root tasks $R = \{T_1, T_2, \ldots, T_u\}$ where $u \leq k$.
- F is the functional description of a complex application.
- Q is the current execution profile.
- ECN_i, $1 \leq i \leq k$, be the flag to indicate the task T_i execution completion notification.
- Random variate x_{ij} where $1 \leq i \leq k$, $1 \leq j \leq n$ is generated from three possible distributions: exponential, normal and uniform. The values in matrix B are used according to the distribution function selected. Uniform and normal distributions need a second value that is provided by user. If $B1$ denotes the first matrix given as the execution matrix B then $B2$ denotes the second matrix provided by user for uniform and normal distributions. Exponential distribution uses $b1_{ij}$ as average execution time. Normal distribution uses $b1_{ij}$ as average execution time and the matrix $B2$ as standard deviation ρ_{ij}. Uniform distribution uses matrix $B1$ as starting point $b1_{ij}$ and $B2$ as ending point $b2_{ij}$. Mean is calculated as $(b1_{ij} + b2_{ij})/2$. A pseudo-random number u is generated from U(0,1). Random variate x_{ij} for each distribution as reported in paper [2] is as follows:

    1. Exponential distribution, E($b1_{ij}$):

    $$x_{ij} = - b1_j \times ln(u)$$

    2. Normal distribution, N($b1_{ij}$, $b2_{ij}^2$):

    $$x_{ij} = \left(\sum_{a=1}^{a=12} u_a - 6\right) \times b2_{ij} + b1_{ij}$$

    3. Uniform distribution, U($b1_{ij}$, $b2_{ij}$):

$$x_{ij} = u \text{ x } (b2_{ij} - b1_{ij}) + b1_{ij}$$

5.2 Overview of Simulation Model

Fig. 5.1 shows the overview of the simulation model used in this thesis. Simulation model has three modules:

1. Task Manager module (TM)
2. Meta Scheduler module (MS)
3. Simulation Engine module (SE)

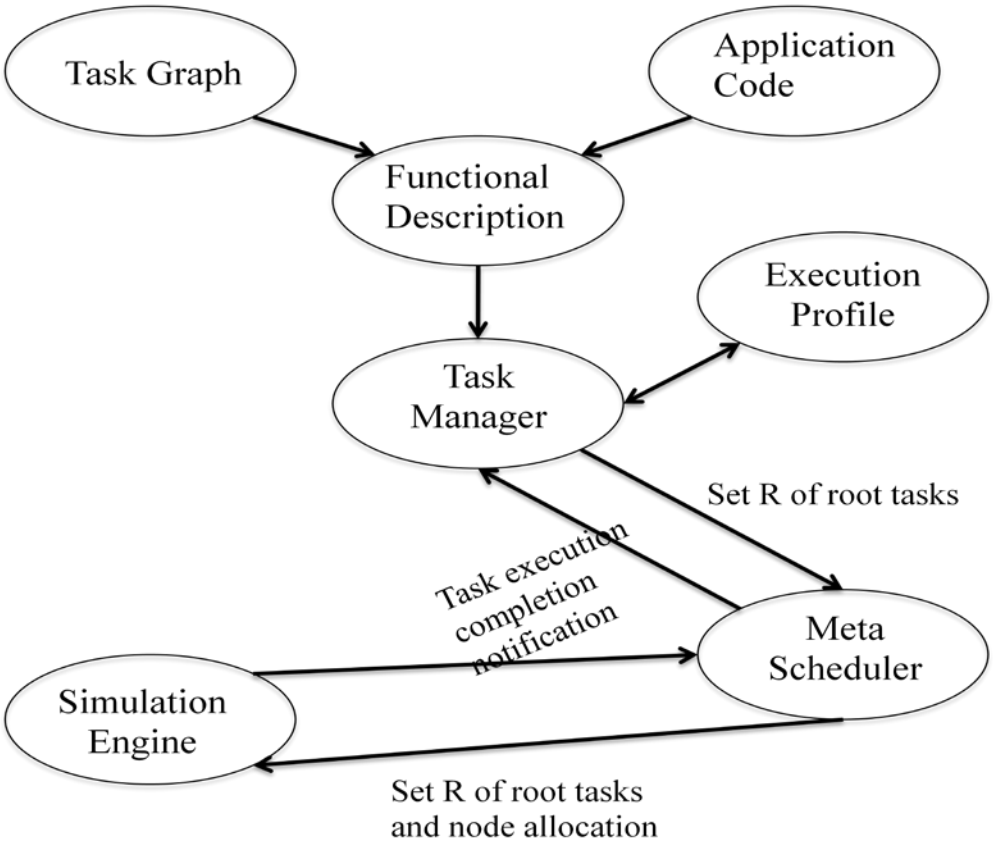

Fig. 5.1 Overview of the simulation model

This simulation model takes functional description of a complex application as input and outputs the total time required for the execution of the complex application. The functional description can be specified from task graph model of the application, or it can be used to reflect the code organization of the application, particularly where functional programming languages are used.

5.2.1 Task Manager Module

Task Manager reads the functional description of a complex application as input and creates an execution profile from the functional description. Initially the execution profile will be the same as the functional description. The TM identifies a set of root tasks *R* from the current execution profile and notifies the MS the set *R*. The TM receives the task execution completion notification from the MS and updates the current execution profile accordingly by removing the completed tasks. The TM stops when there are no more root tasks to schedule. The algorithm for the TM is as follows:

Algorithm: TM

1. TM uses the functional description F as initial execution profile Q
2. TM identifies a set of root tasks R from Q
3. TM notifies MS about R to schedule tasks that are not yet scheduled
4. TM waits for execution completion notification ECN_i from MS
5. TM updates Q and R by removing the task T_i that completed execution
6. Repeat steps (2 – 5) as long as $R \neq \varnothing$

5.2.2. Meta Scheduler Module

Meta Scheduler has all the information about nodes in the grid environment. The MS receives the set *R* from the TM. The MS schedules tasks in *R* on particular nodes based on some task scheduling heuristic scheme, which is detailed in Chapter 6. The algorithm for the MS is as follows:

Algorithm: MS

1. MS receives *R* from TM
2. MS schedules tasks in *R* that are not yet scheduled on some node W_j by using any task scheduling heuristic scheme and notifies SE about *R* and the task allocation details
3. MS waits for ECN_i and the time taken by task T_i to complete execution from SE
4. MS notifies TM about ECN_i

5.2.3. Simulation Engine Module

Simulation Engine (SE) receives the notification from the MS about the set of root tasks *R* to execute. Notification from the MS includes details about which task to execute on which node in the grid network. The SE executes the task and also records the total time taken by the task to complete execution. The SE then notifies the MS about the task execution completion. The algorithm for the SE is as follows:

Algorithm: SE

1. SE receives R and the task allocation details from MS
2. SE executes tasks in R that are not yet scheduled on some node W_j
3. SE notifies MS about ECN_i and the time taken by task T_i to complete execution

Chapter 6

Task Scheduling Heuristic Schemes

In order to serve the increasing needs of large applications, people try to provide high performance computation power to a single machine by increasing its capacity or constructing a distributed system with scalable set of machines [22]. Scheduling or allocating a task to a particular node in a distributed environment is a NP-hard problem. Many task scheduling heuristic schemes are proposed to address this issue. In this thesis, two task scheduling heuristic schemes are implemented: Shortest Estimated Execution Time First (SEETF) [23, 24, 25] and Minimum Data Packets and Communication Cost (MDPCC) [6].

The task scheduling algorithms used in this thesis are all static and heuristic; the decisions are made before the execution (*static*), and there are definite rules in the scheduling procedure for task allocations (*heuristic*). A task allocation heuristic scheme generates the task allocation matrix A_{kxm} used in the simulation model. The following section explains the two task scheduling heuristic schemes used in this thesis:

6.1 Shortest Estimated Execution Time First (SEETF)

In this scheme [3, 23, 24, 25], the task T_i is selected from the task set T and assigned to the node W_j in the grid network that executes the T_i faster.

The elements of task allocation matrix from the SEETF algorithm are determined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } b_{ij} = \min_j\{b_{ij}\} \\ 0 & Otherwise \end{cases} \quad (6.1)$$

where $1 \leq i \leq k$, $1 \leq j \leq n$, k is the number of tasks, n is the number of nodes, $a_{ij} = 1$ if task T_i has been allocated to node W_j and 0 otherwise, b_{ij} is the average execution time of task T_i on node W_j

6.2 Minimum Data Packets and Communication Cost (MDPCC)

This heuristic takes into account the data packet cost between tasks, the communication cost between nodes and the execution time of tasks on nodes in the grid network. Here, the first task T_1 is executed on a node W_r that has the minimum average communication cost. The remaining tasks T_i will be executed on a node W_j that has the minimum of the combination of communication cost from its predecessor task's node W_r, the data packet cost and the execution time of task T_i on node W_j**.** The elements of task allocation matrix for MDPCC are determined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } \min_j\{c_{rj} \text{ x } data\_packet + b_{ij}\} \\ 0 & Otherwise \end{cases} \quad (6.2)$$

where $2 \leq i \leq k$, $1 \leq j \leq n$, $1 \leq r \leq n$, k is the number of tasks , n is the number of nodes, *data_packet* is the data packet dependency between current task and its predecessor task, a_{ij} = 1 if task T_i has been allocated to node W_j and 0 otherwise, b_{ij} is the average execution time of task T_i on node W_j, c_{rj} is the average communication cost to transfer a data packet of standard size from node W_r to node W_j

Chapter 7

Implementation

This chapter outlines the implementation details of the simulation model reported in Chapter 5. There are two implementations and they are as follows:

1. Implementation I
2. Implementation II

Implementation I exactly depict the simulation model shown in Fig. 5.1. This implementation includes all the three modules – the Task Manager module, the Meta Scheduler module and the Simulation Engine module. But the Implementation II consists of only two modules – the Meta Scheduler module and the Simulation Engine module. In Implementation II, the Task Manager module is bypassed and instead of that the Simulation Engine module uses the full specification of the task graph. SimGrid internally does the task management. Implementation II is mainly used to validate results obtained from Implementation I. These two implementations can be used to evaluate a task graph in different types of network as reported in paper [2].

Both the implementations are done in C programming language. The SimGrid software is used for building the simulation model. Different text files are used for the implementation purpose. The input files are functional_description.txt, meta_scheduler_3n.txt, and node_allocation.txt. The functional description of a complex

application is given in the file functional_description.txt. It contains the number of paths in the functional description, the maximum number of tasks in a path and the functional description of the complex application. The file meta_scheduler_3n.txt is the meta scheduler input file, which contains the number of tasks, the number of nodes, the data packet matrix Pkt_{kxk} , the execution time matrix B_{kxn}, and the node communication cost matrix C_{nxn}. The file node_allocation.txt contains information about each task like the task name, the node on which the task should execute and the random variate of the execution time of that task. The output file contains the total execution time of the complex application for each simulation cycle. The file, execution_times_app_tm.txt, is the output file for the Implementation I and the file, execution_times_app.txt, is the output file for the Implementation II. The contents of the output file should be deleted for each run of the experiment. The file 3nodes.xml is the platform file used by the SimGrid program that contains information about the network topology. The following section explains the two implementations.

7.1 Implementation I

This implementation contains all the three modules and explains our idea of the functional description, the execution profile and the set of root tasks. The C source code files used for this implementation are Imp1_SimGrid.c and Imp1_Main.c. The following section shows the mapping of three modules used in this implementation:

7.1.1 Task Manager module

The procedures tm_read_func_desc(), tm_notify_ms() and update_completed_tasks() in Imp1_SimGrid.c do the functions of the TM. The procedure tm_read_func_desc() reads the functional description of the complex application from the file functional_description.txt and creates a current execution profile, which is a matrix of tasks. The procedure tm_notify_ms() notifies the MS about the set of root tasks R by using the *current_schedule* structure to schedule tasks, which are not yet scheduled. The procedure update_completed_tasks() updates the *completed_tasks* structure when a task completes execution.

7.1.2 Meta Scheduler module

The procedures meta_scheduler_schedule(), update_completed_tasks() in Imp1_SimGrid.c and read_input_file(), calc_exec_times(), change_exectimes(), write_to_file() in Imp1_Main.c do the functions of the MS. The procedure read_input_file() reads the input file meta_scheduler_3n.txt that has information about the number of tasks, the number of nodes, the data packet matrix, the execution time matrix, and the node communication cost matrix. The function read_input_file() prompts the user to enter the distribution type and the scheduling heuristic scheme type. Based on the scheduling heuristic scheme selected, the heuristic function either seetf() or mdpcc() do the node allocation for each task. The procedure calc_exec_times() calculates the random variates for the selected distributed by using random_exponential() or

random_uniform() or random_normal() procedure. The procedure change_exectimes() updates the random variates of execution time in the node allocation array. The procedure write_to_file() writes the node allocation array into the file node_allocation.txt. The procedure meta_scheduler_schedule() schedules the task on the respective node in the network by using the node allocation structure node_allocation.

Users can develop their own code to implement any new task scheduling heuristic scheme. Task allocation is done in Imp1_Main.c using user given average execution times *exec_pdf[n][k].pdf_mean* where *n* is the number of nodes, *k* is the number of tasks. Perform the necessary calculation and updates the allocation matrix *allocate[k][2]*. The element *allocate[j][0]*, $1 \leq j \leq k$ contains the node number W_i, $1 \leq i \leq n$ on which the task T_j should execute and the element *allocate[j][1]* contains the random variate of the average execution time *exec_times[x][y]*, $1 \leq x \leq n$, $1 \leq y \leq k$ of the task T_j on the node W_i. The function change_exectimes() updates the allocation matrix element *allocate[j][1]*. To integrate the new heuristic code into the simulation model the user should do the following steps:

1. Write a new heuristic function in Imp1_Main.c
2. Use average execution time matrix *exec_pdf[n][k].pdf_mean*, the nodes communication cost matrix *nodes_comm[n][n]*, the task data dependency matrix *task_data[k][k]* to do the necessary calculation

3. Update the allocation matrix element *allocate[j][0]* with the allocated node number for each task T_j

7.1.3 Simulation Engine module

The procedures simulate() in Imp1_Main.c and SD_simulate() in Imp1_SimGrid.c do the SE functions. The procedure simulate() in Impl_Main.c calls the Imp1_SimGrid.c program for each simulation cycle. The procedure SD_simulate() in Imp1_SimGrid.c performs the simulation by executing tasks on nodes in the network. The function SD_task_watch() is used to pause the simulation when a task completes execution.

The output of Imp1_Simgrid.c is the total execution time of the complex application and it is written to the file execution_times_app_tm.txt. The simulation cycle is repeated for 1000 times. The output file is used for analysis purpose and to plot various distribution graphs.

7.2 Implementation II

In this implementation, the TM module is bypassed and instead of that the Simulation Engine module uses the full specification of the task graph. The full specification is made available by adding the task dependencies. Task dependencies are added using the SimGrid function SD_task_dependency_add reported in section 4.5.9. SimGrid internally does the task management. Neither the functional description nor the

execution profile is used in this implementation. The C source code files used for this implementation are Imp2_SimGrid.c and Imp2_Main.c. The following section shows the mapping of two modules used in this implementation:

7.2.1 Meta Scheduler module

The procedures read_input_file(), calc_exec_times(), change_exectimes(), write_to_file() in Imp2_Main.c do the functions of the MS. The procedure read_input_file() reads the input file meta_scheduler_3n.txt that has information about the number of tasks, the number of nodes, the data packet matrix, the execution time matrix, and the node communication cost matrix. The function read_input_file()prompts the user to enter the distribution type and the scheduling heuristic type. Based on the scheduling heuristic selected, the heuristic function either seetf() or mdpcc() do the node allocation for each task. The procedure calc_exec_times() calculates the random variates for the selected distributed by using random_exponential() or random_uniform() or random_normal() procedure. The procedure change_exectimes() updates the random variates of the execution time in the node allocation array for each simulation cycle. The procedure write_to_file() writes the node allocation array into the file node_allocation.txt, which is used by the simulation engine. The SimGrid function SD_task_schedule reported in section 4.5.5 is used to schedule the tasks.

Users can develop their own code to implement any new task scheduling heuristic scheme. Task allocation is done in Imp2_Main.c using user given average execution

times *exec_pdf[n][k].pdf_mean* where *n* is the number of nodes, *k* is the number of tasks. Perform the necessary calculation and update the allocation matrix *allocate[k][2]*. The element *allocate[j][0], 1≤j≤k* contains the node number W_i, $1 \le i \le n$ on which the task T_j should execute and the element *allocate[j][1]* contains the random variate of the average execution time *exec_times[x][y]*, $1 \le x \le n$, $1 \le y \le k$ of the task T_j on the node W_i. The function change_exectimes() updates the allocation matrix element *allocate[j][1]*. To integrate the new heuristic code into the Simulation model the user should do the following steps:

1. Write a new heuristic function in Imp2_Main.c
2. Use average execution time matrix *exec_pdf[n][k].pdf_mean*, the nodes communication cost matrix *nodes_comm[n][n]*, the task data dependency matrix *task_data[k][k]* to do the necessary calculation
3. Update the allocation matrix element *allocate[j][0]* with the allocated node number for each task T_j

7.2.2 Simulation Engine module

The procedures simulate() in Imp2_Main.c and SD_simulate() in Imp2_SimGrid.c do the SE functions. The procedure simulate() in Imp2_Main.c calls the Imp2_SimGrid.c program for each simulation cycle. The procedure SD_simulate() in Imp2_SimGrid.c performs the simulation by executing the tasks on the nodes in the network.

The output of Imp2_Simgrid.c is the execution time of the complex application and it is written to the file execution_times_app.txt. The simulation cycle is repeated for 1000 times. The output file is used for analysis purpose and to plot various distribution graphs.

7.3 Assumptions and Restrictions

In the software implementation of the simulation model, some assumptions and restrictions are made. The number of nodes in the system is limited to 100. The number of tasks in the complex application is limited to 1000. A separate pdf is used for each task T_i, $1 \leq i \leq k$ allocated to node W_j, $1 \leq j \leq n$. Execution times can be exponentially, normally or uniformly distributed. The nodes in the grid network are assumed to be homogeneous. The paths in the functional description of the complex application should be of same length. If paths are not of same length, insert zero in the path that has no task in that level.

Chapter 8

Experimentation and Analysis

If the proposed simulation model evaluates the performance of a complex application correctly, then the result of the simulation model should be the same as the result from a numerical method. To validate the simulation model, a simple application represented by a task graph U shown in Fig. 8.1 is analyzed in this chapter. One more validation is performed on the simulation model by comparing the results obtained from Implementations I and II.

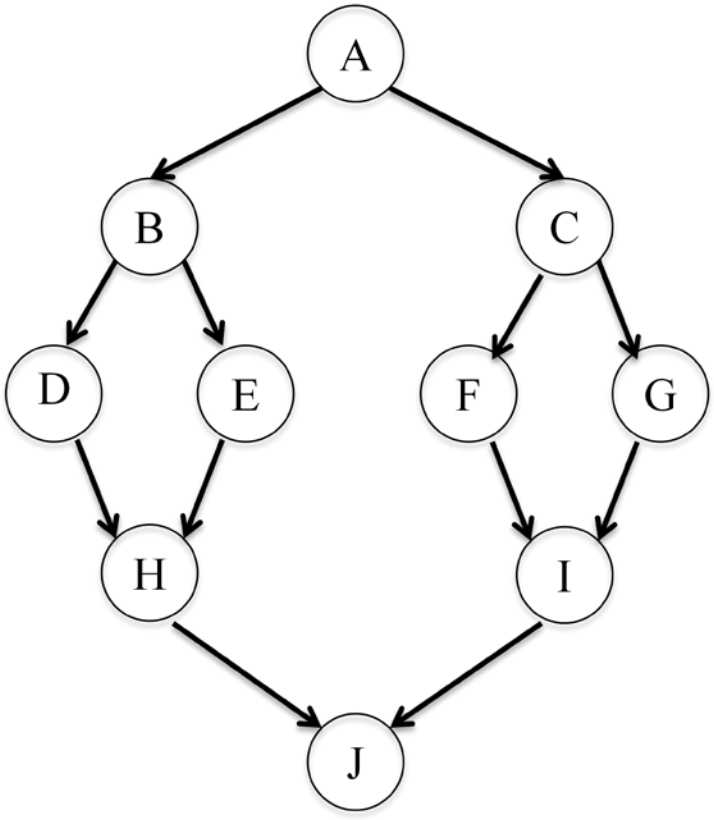

Fig. 8.1 Task graph U

The parameters used are as follows:

- Number of nodes, n=3
- Number of tasks, k=10
- A k x n execution time matrix B_{ij}, $1 \leq i \leq k$, $1 \leq j \leq n$ where b_{ij} is the average execution time of T_i on W_j:

$$B = \begin{pmatrix} .2 & .4 & .4 \\ .5 & .7 & 1 \\ .3 & 1 & .4 \\ 1 & .2 & .7 \\ .6 & 1 & .6 \\ .3 & .5 & .7 \\ 1 & 1 & .8 \\ .4 & .1 & .5 \\ .9 & .6 & 1 \\ .4 & 1 & .3 \end{pmatrix}$$

- A n x n communication cost matrix C_{rs}, $1 \leq r, s \leq n$ where each entry c_{rs} is the average communication cost to transfer a data packet of standard size from W_r to W_s:

$$C = \begin{pmatrix} 0 & 50 & 100 \\ 200 & 0 & 50 \\ 50 & 100 & 0 \end{pmatrix}$$

- A k x k task data dependency matrix Pkt_{ij}, $1 \leq i, j \leq k$ where $pkt[i, j]$ is the average number of data packets of standard size that is sent from T_i to T_j.

$$Pkt = \begin{pmatrix} 0 & 5 & 10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 20 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 20 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 15 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The execution times, the communication costs and the data dependencies between tasks are obtained from a user given meta_scheduler_3n.txt file. The functional description of the task graph U in Fig 8.1 is shown below:

U[J[H[D[B[A]], E[B[A]]], I[F[C[A]], G[C[A]]]]] (8.1)

From equation (8.1), the following set of queue structures Q is obtained. The set Q is then used as an execution profile. Each q_i in Q represents a path in the Fig 8.1.

$$Q = \begin{pmatrix} A\ B\ D\ H\ J \\ A\ B\ E\ H\ J \\ A\ C\ F\ I\ J \\ A\ C\ G\ I\ J \end{pmatrix}$$

Thus, the simulation model uses the functional description to create an execution profile, which will be updated when each task completes execution. The simulation model outputs the total execution time of the application. The following comparisons are made for the exponential distribution of execution times. Table 8.1 shows the comparison of Mean of the Total Execution Time (MTET) of the application for different heuristics in Implementations I and II. Table 8.2 shows the comparison of 99% confidence intervals of MTET.

Table 8.1 Comparison of MTET from Implementations I and II

Heuristic	Implementation I MTET in seconds	Implementation II MTET in seconds
Shortest Estimated Execution Time First	3.562761	3.528183
Minimum Data Packets and Communication Cost	6.574534	6.490911

Table 8.2 Comparison of 99% confidence intervals of MTET from Implementations I and II

Heuristic	Implementation I 99% confidence interval	Implementation II 99% confidence interval
Shortest Estimated Execution Time First	(3.521344, 3.604179)	(3.486737, 3.569630)
Minimum Data Packets and Communication Cost	(6.504618, 6.644451)	(6.421139, 6.560682)

Table 8.1 and Table 8.2 show the comparison of MTET obtained from two implementations. Implementation I integrate a task management scheme and the libraries provided by SimGrid, where as in the Implementation II a full specification of the task graph is made available to the SimGrid. As reported by SimGrid simulation tool, the MTET obtained from two implementations are approximately equal. This provides a validation procedure for the proposed simulation model.

A second validation is a numerical method that is conducted on a consistent run in a single trial for the task graph U using the execution time of tasks reported by SimGrid. The execution time of the task graph U is the maximum of the sum of the execution time of tasks in various paths in Q. This can be reported as follows:

$$Execution\ Time_U = \max_i\{P_i\} \tag{8.2}$$

where $1 \leq i \leq n$, n is the number of paths in Q. P_i corresponds to the execution time of the path q_i.

Table 8.3 shows the execution time of various tasks in Implementations I. Table 8.4 shows the sum of the execution time of tasks in various paths in Q. The total execution time for entire task graph U as reported by SimGrid is 2.800037 seconds in Implementation I. We can see from Table 8.4 that the path ACGIJ takes the maximum execution time and is equal to 2.800037 seconds, which is the same as the total execution time for the task graph U reported by SimGrid. This provides the second validation.

Table 8.3 Execution time of various tasks in Implementations I

Task	Execution Time in seconds
A	0.372817
B	0.880002
C	0.837749
D	0.165647
E	0.125623
F	0.188108
G	1.357239
H	0.021799
I	0.127695
J	0.104537

Table 8.4 Execution time for each path in Q

Path q_i	Execution Time of q_i
ABDHJ	1.544802
ABEHJ	1.610674
ACFIJ	1.630906
ACGIJ	2.800037

The following figures show the Cumulative Distribution Function (CDF) of the execution time of the task graph U at time t. The CDF gives the complete probability distribution of a random variable. The graphical representation in Fig 8.2, Fig 8.3 shows the total execution time obtained for the probability of completion at time t, $P(x \leq t)$ of the application. We can observe from the graphs that the SEETF heuristic performs better than MDPCC heuristic in both exponential and normal distribution of execution times. The graphs show how different task scheduling heuristic scheme affect the total execution time of the application. The results reported compare the performance of a complex application under different scheduling heuristic schemes.

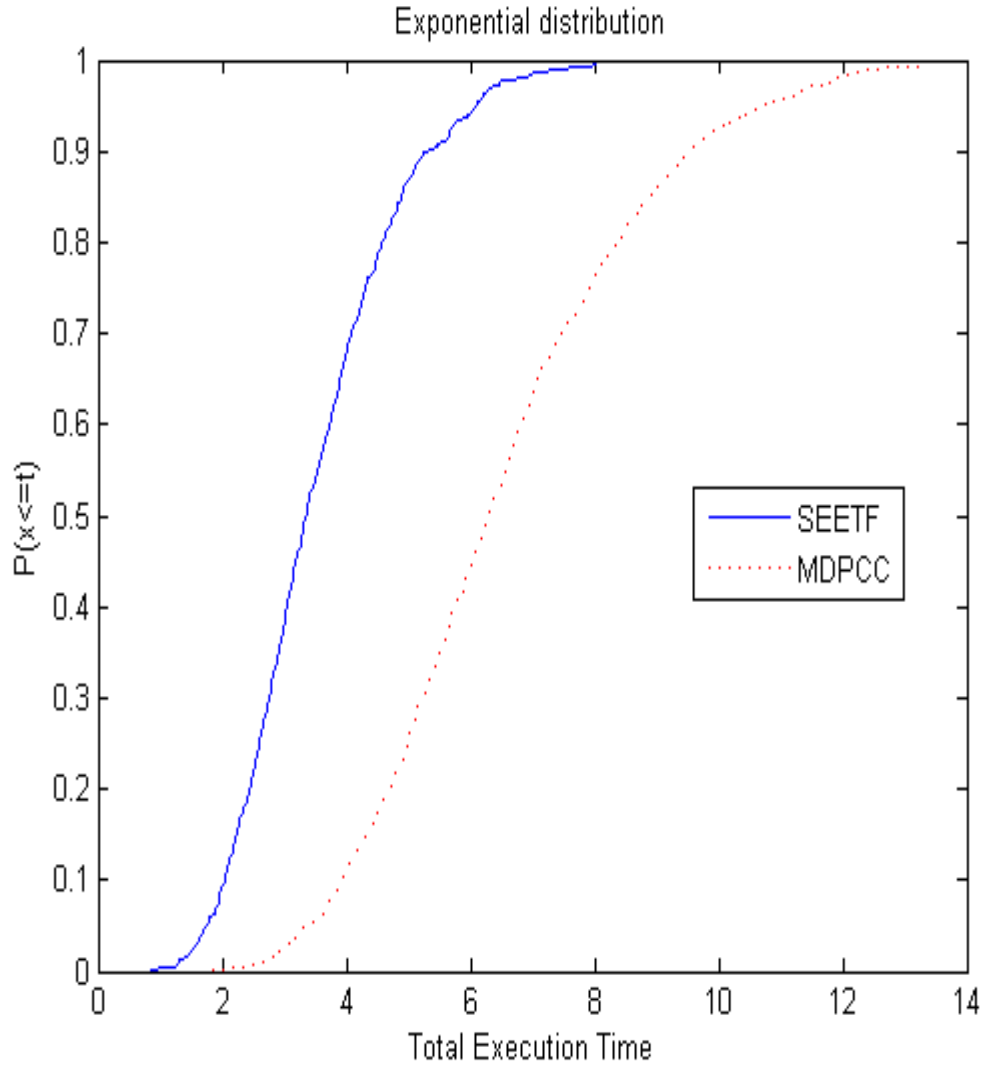

Fig. 8.2 CDF of the execution time of a task graph U at time t for exponential distribution in Implementation I.

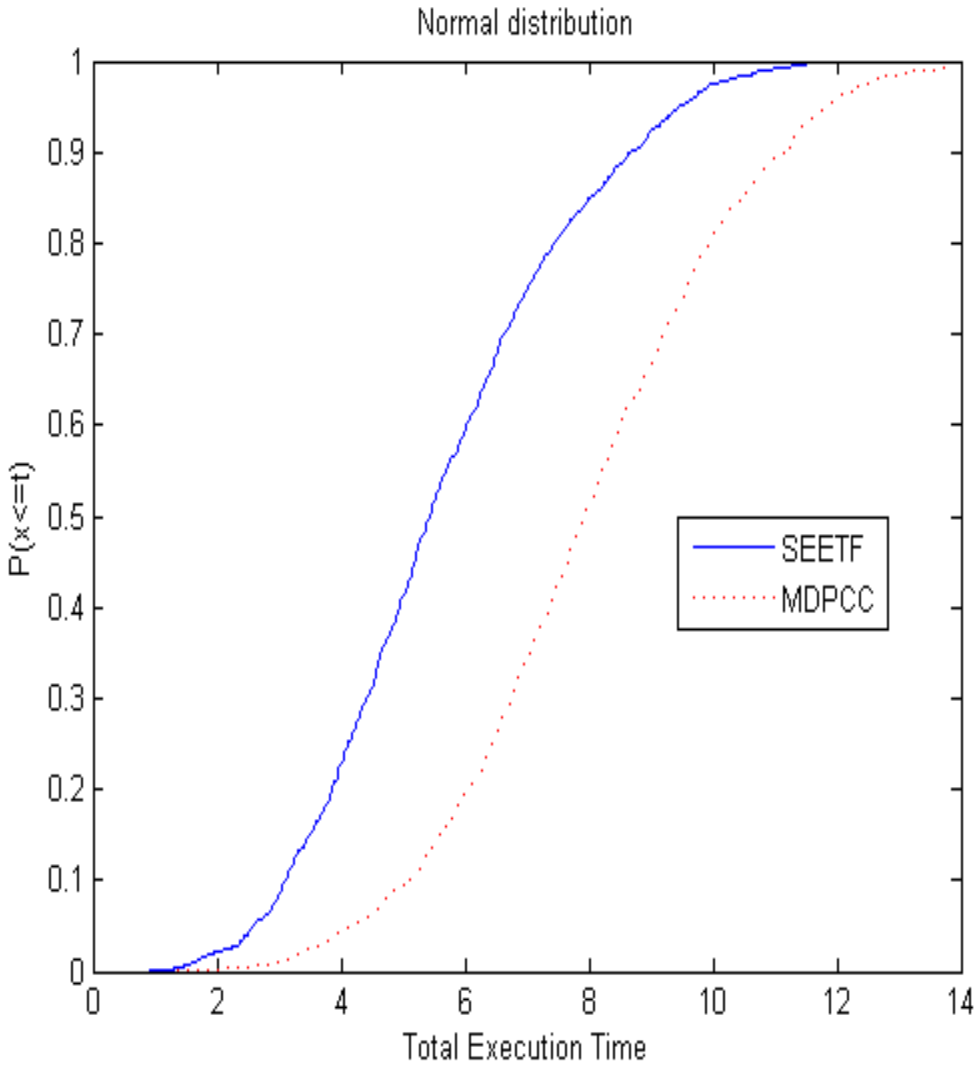

Fig. 8.3 CDF of the execution time of a task graph U at time t for normal distribution in Implementation I.

Chapter 9

Conclusion and Future Work

The proposed simulation model integrates a task management scheme and the libraries provided by the SimGrid simulation tool to evaluate the performance of a complex application submitted for execution to a grid-computing environment under a given scheduling heuristic scheme. The task management scheme uses an alternative representation to the task graph model referred to as the functional description of the application. The functional description represents the initial execution profile of an application submitted to the grid for execution. The execution profile becomes dynamic in nature as the application progresses. The root tasks are identified from the execution profile and submitted to a grid for execution. These root tasks indicate that all precedent tasks have already been executed, removing any data dependencies between tasks. Only the root tasks are updated in response to execution completion notifications from the grid. Simulation of grid applications allows the usage of several types of distributions. In this thesis, we have implemented normal, uniform, and exponential distribution. Using SimGrid as the simulation engine for the model proposed the evaluation of task scheduling heuristics is possible. The simulation itself is made simpler because all task dependencies are taken care by the task manager module. All the results are based on exponential distribution of execution times. The results reported compare the performance of a complex application under different scheduling heuristic schemes.

From the analysis, we observed that SEETF heuristic performs better than MDPCC heuristic in exponential and normal distribution.

Future work can include additional task scheduling heuristic schemes. Because of the nature of simulation procedure, more complex dynamics can be easily incorporated, such as dynamic allocation. Getting the functional description directly from functional programming language is not covered in this thesis. Also, the analysis using simulation can be extended to determine an optimal size of the network in terms of number of nodes to achieve the best performance of a given grid application.

Bibliography

[1] N. Lopez-Benitez, and P. Andersen, "Dynamic Structures for the Management of Complex Applications in Grid Environments," in *Proceedings of the 2009 International Conference on Grid Computing & Applications*, Las Vegas, Nevada, USA, 2009, pp. 80-85.

[2] N. Lopez-Benitez, and J.-Y. Hyon, "Simulation of Task Graph Systems in Heterogeneous Computing Environments," in *Proceedings of the Eighth Heterogeneous Computing Workshop*, Lubbock, 1999, pp. 112.

[3] J.-Y. Hyon, "Simulation of Task Graph Systems using Stochastic Petri Net Models," Computer Science, Texas Tech University, Lubbock, 1998.

[4] K. Thulasiraman, and M. Swamy, *Graphs: Theory and Algorithms*, New York: A Wiley-Interscience Publication, 1992.

[5] A. Miller, "The Task Graph Pattern," in *Second Annual Conference on Parallel Programming Patterns (ParaPLoP)*, Carefree, AZ, United States, 2010.

[6] S. Aditham, "Task Management Driven Simulation of Grid Applications Using Stochastic Petri Nets," Computer Science, Texas Tech University, Lubbock, 2010.

[7] R. Rubinstein, *Simulation and The Monte Carlo Method*, New York: John Wiley & Sons, 1981.

[8] G. Fishman, *Monte Carlo: Concepts, Algorithms, and Applications*, New York: Springer, 1996.

[9] R. Rubinstein, *Monte Carlo Optimization, Simulation and Sensitivity of Queuing Networks*, New York: John Wiley & Sons, 1986.

[10] M. Hamburg, *Statistical Analysis For Decision Making*, New York: Harcourt, Brace & World, 1970.

[11] K. S. Vallerio, and N. K. Jha, "Task Graph Extraction for Embedded System Synthesis," in *Proceedings of the Sixteenth International Conference on VLSI Design*, New Delhi, India, 2003, pp. 480.

[12] M. Girkar, and C. D. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 166-178, 1992.

[13] A. Bui, O. Flauzac, and C. Rabat, "Fully Distributed Active and Passive Task Management for Grid Computing," in *Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, Hagenberg, 2007, pp. 21.

[14] B. Alain, "Fully Distributed and Fault Tolerant Task Management Based on Diffusions," in *Seventeenth Euromicro International Conference on Parallel, Distributed, and Network-Based Processing PDP*, France, 2009, pp. 355-360.

[15] P. A. Fishwick, "SimPack: getting started with simulation programming in C and C++," in *Proceedings of the Twenty-fourth Winter Simulation Conference*, Arlington, Virginia, United States, 1992, pp. 154-162.

[16] F. Gomes, S. Franks, B. Unger *et al.*, "SimKit: a high performance logical process simulation class library in C++," in *Proceedings of the Winter Simulation Conference*, Arlington, Virginia, United States, 1995, pp. 706-713.

[17] F. Howell, and R. McNab, "SimJava: A Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling," in *Proceedings of the First International Conference on Web-based Modelling and Simulation*, San Diego, California, United states, 1998.

[18] A. Legrand, L. Marchal, and H. Casanova, "Scheduling Distributed Applications: the SimGrid Simulation Framework," in *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid*, Washington, DC, United States, 2003, pp. 138.

[19] H. J. Song, X. Liu, D. Jakobsen *et al.*, "The MicroGrid: A scientific tool for modeling Computational Grids," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM) (Supercomputing '00)*, Washington, DC, United States, 2000.

[20] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: A Generic Framework for Large-Scale Distributed Experiments," in *Proceedings of the Tenth International Conference on Computer Modelling and Simulation*, Washington, DC, United States, 2008, pp. 126-131.

[21] "Welcome to the SimGrid project!." Retrieved 22 October 2010, from http://simgrid.gforge.inria.fr/

[22] W. Ming, and S. Xian-He, "A General Self-Adaptive Task Scheduling System for Non-Dedicated Heterogeneous Computing," in *Fifth IEEE International Conference on Cluster Computing (CLUSTER'03)*, Hong Kong, 2003, pp. 354.

[23] S. C. S. Porto, and D. A. Menasce, "Processor assignment in heterogeneous message passing parallel architectures," in *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sceince*, Kauai, Hawaii, 1993, pp. 496-505.

[24] S. C. S. Porto, "Heuristic Scheduling Algorithms for Task Scheduling in Heterogeneous Multiprocessor Architectures," Departamento de Informatica, PUCRIO, Brazil, 1991.

[25] S. Porto, and S. Tripathi, "Static Heuristic Processor Assignment in Heterogeneous Multiprocessors," *International Journal of High Speed Computing*, pp. 115 - 137, 1994.

Appendix

Source Code of Simulation Model

Implementation I

```
/*********************************************************************
** FILE NAME: Imp1_Main.c
** AUTHOR: UDAY KUMAR B NARAYANAPPA
** DATE:  11/01/2010
** DESCRIPTION: gets the probability of execution time of each task on
**              on each node, communication costs between nodes, and
**              allocates node to tasks and then generate random
**              numbers and simulates the complex application.
**  OUTPUT: 1000 instances of total execution time and MTET values
*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include "simdag/simdag.h"
#include "xbt/ex.h"
#include "xbt/log.h"

#define LINE_LENGTH 200
#define MAX_TASKS 1000  // maximum size of task graph
#define MAX_NODES 100   // maximum size of grid network
#define NUM_ITERATION 1000 // number of simulation cycle

int num_tasks;                // number of tasks limited by MAX_TASKS
int num_nodes;                // number of nodes limited by MAX_NODES
char distribution;          // distribution type
char heu;                     // heuristic type
float exec_times[MAX_NODES][MAX_TASKS]; // execution time of tasks on
different nodes
float allocate[MAX_TASKS][2]; // allocation matrix
int node_comm[MAX_NODES][MAX_NODES]; // nodes communication cost
int adjacency_matrix[MAX_TASKS][MAX_TASKS]; // adjacency matrix of task
graph
int task_data[MAX_TASKS][MAX_TASKS]; // task data dependency
float avg_node_comm[MAX_NODES]; // avg. node communication
float tet[NUM_ITERATION]; // total execution time

FILE *inp_f, *out_f;

typedef struct{
    float pdf_mean;
    float pdf_variance;
} PDF;
```

```
PDF exec_pdf[MAX_NODES][MAX_TASKS]; // the matrix for execution times
pdf

void read_input_file(void);
void calc_exec_times(void);
void seetf(void);
void mdpcc(void);
void change_exectimes(void);
void write_to_file(void);
int minim(int task_dependency[MAX_TASKS][2], int count);
float random_exponential(float mean);
float random_uniform(float begin, float end);
float random_normal(float mean, float std);
void analysis(void);
void simulate(void);

int main()
{
    int i, k;

    // read the meta scheduler file
    read_input_file();

    // user selected heuristic
    if(heu == 's')
      seetf();
    else if(heu == 'm')
        mdpcc();

    // experiment is repeated for NUM_ITERATION iterations
    for(k=1;k<=NUM_ITERATION;k++)
    {

      // generate random variates of selected distribution
      calc_exec_times();

      // update the allocation matrix with generated random variates
      change_exectimes();

      // print the allocation matrix
      printf("\n \n");
      for(i = 0; i < num_tasks; i++)
            printf("\n%d %f\t",(int)allocate[i][0],allocate[i][1]);
      printf("\n \n");

      // write the allocation matrix into node_allocation.txt file
      write_to_file();

      // call the SimGrid program to start the simulation
      simulate();

    }
```

```
    // calculate mean of total execution time(MTET) and its 99%
confidence interval values
    analysis();

    return 1;

}

/*********************************************************************/
/*********************************************************************/
/*  read_input_file()                                                */
/*********************************************************************/
/*********************************************************************/

void read_input_file()
{
    char
ipfilename[LINE_LENGTH]="/Users/Uday/Desktop/Files/meta_scheduler_3n.tx
t";
    int i,j;

    inp_f = fopen(ipfilename,"r");

    // get distribution type
    printf("\n Choose the type of Distribution.\n Enter 'n' for Normal,
'u' for Uniform, 'e' for Exponential\n");
    scanf("%s",&distribution);

    //get heuristic type
    printf("\n Choose the Heuristic.\n Enter 's' for Shortest Estimated
Execution Time First, 'm' for Minimum Data Packets and Communication
Cost\n");
    scanf("%s",&heu);

    // read the number of tasks
    fscanf(inp_f, "%d", &num_tasks);

    // read the number of nodes
    fscanf(inp_f, "%d", &num_nodes);

    // create the adjacency matrix
    for(i = 0; i < num_tasks; i++)
      for(j = 0; j < num_tasks; j++)
        {
            fscanf(inp_f, "%d", &task_data[i][j]);
            if(task_data[i][j]!=0)
                  adjacency_matrix[i][j]=1;
            else
                  adjacency_matrix[i][j]=0;
      }

    // get the means of execution time
```

```
    for(i = 0; i < num_nodes; i++)
      for(j = 0; j < num_tasks; j++)
            fscanf(inp_f, "%f", &exec_pdf[i][j].pdf_mean);

    // get the variances
    for(i = 0; i < num_nodes; i++)
      for(j = 0; j < num_tasks; j++)
            fscanf(inp_f, "%f", &exec_pdf[i][j].pdf_variance);

    // get the nodes communication cost
    for(i = 0; i < num_nodes; i++)
      for(j = 0; j < num_nodes; j++)
            fscanf(inp_f, "%d", &node_comm[i][j]);


    fclose(inp_f);

}


/*********************************************************************/
/*********************************************************************/
/*  calc_exec_times()                                                */
/*********************************************************************/
/*********************************************************************/

void calc_exec_times()
{
    int i,j;

    // generate exponential distribution random variates
    if(distribution == 'e')
    {
      printf("exponential\n");
      for(i = 0; i < num_nodes; i++)
            for(j = 0; j < num_tasks; j++)

    exec_times[i][j]=random_exponential(exec_pdf[i][j].pdf_mean);
    }

    // generate normal distribution random variates
    else if(distribution == 'n')
    {
      printf("normal\n");
        for(i = 0; i < num_nodes; i++)
            for(j = 0; j < num_tasks; j++)

    exec_times[i][j]=random_normal(exec_pdf[i][j].pdf_mean,exec_pdf[i][
j].pdf_variance);
    }

    // generate uniform distribution random variates
    else if(distribution == 'u')
```

```
    {
      printf("uniform\n");
      for(i = 0; i < num_nodes; i++)
            for(j = 0; j < num_tasks; j++)

    exec_times[i][j]=random_uniform(exec_pdf[i][j].pdf_mean,exec_pdf[i]
[j].pdf_variance);
    }
}

/*********************************************************************/
/*********************************************************************/
/*  random_uniform(mean, variance)                                   */
/*********************************************************************/
/*********************************************************************/

float random_uniform(float begin, float end)
{
    float ran_num;
    float mean, width, diff;

    width = end - begin;
    if(width < 0)
      printf("\nINVALID INPUT MEAN, VARIANCE\n");
    mean = (begin + end) / 2;
    diff = mean - (width / 2);

    ran_num = rand() / (float) RAND_MAX;
    ran_num = ran_num * width;
    ran_num = ran_num + diff;
    ran_num = ran_num + begin;

    return(ran_num);

}


/*********************************************************************/
/*********************************************************************/
/*  random_normal(mean, variance)                                    */
/*********************************************************************/
/*********************************************************************/

float random_normal(float mean, float std)
{

    int i;
    float z, ran_num;

    z = 0;
    for( i = 0; i < 12; i++)
      z = z + (rand() / (float) RAND_MAX);
    z = z - 6;  // N(0,1)
```

```
    ran_num = z * std + mean; // x = z* standard deviation + mean

    return(ran_num);

}

/*********************************************************************/
/*********************************************************************/
/*  random_exponential(mean)                                         */
/*********************************************************************/
/*********************************************************************/

float random_exponential(float mean)
{

    float ran_num;

    ran_num = 0;
    while(ran_num == 0)
        ran_num = rand() / (float) RAND_MAX;

    ran_num = mean * log(ran_num);
    ran_num = - ran_num;

    return(ran_num);

}

/*********************************************************************/
/*********************************************************************/
/*  change_exectimes()                                               */
/*********************************************************************/
/*********************************************************************/

void change_exectimes()
{
    int i,j;
    for(i = 0; i < num_tasks; i++)
    {
      j=(int)allocate[i][0];
      allocate[i][1]=exec_times[j-1][i];
    }
}


/*********************************************************************/
/*********************************************************************/
/*  SEETF heuristic                                                  */
/*********************************************************************/
/*********************************************************************/

void seetf()
```

```
{
    int i,j,pos;
    double min;
    for(i = 0; i < num_tasks; i++)
    {
      min=exec_pdf[0][i].pdf_mean;
      pos=0;
      for(j = 1; j < num_nodes; j++)
      {
            if(exec_pdf[j][i].pdf_mean < min)
            {
                  min=exec_pdf[j][i].pdf_mean;
                  pos=j;
            }
      }
      allocate[i][0]=pos+1;
    }
}


/*********************************************************************/
/*********************************************************************/
/*  MDPCC heuristic                                                  */
/*********************************************************************/
/*********************************************************************/

void mdpcc()
{

    int i,j,pos,pos2,allo_proc1,final_task_allo_node;
    float min,min2,x;

    // calculate average node communication
    for(i = 0; i < num_nodes; i++)
    {
      int sum=0;
      for(j = 0; j < num_nodes; j++)
      {
            sum = sum + node_comm[i][j];
      }
      avg_node_comm[i] = sum / num_nodes;
    }

    // first task is allocated a node which has minimum average node
communication
    min = avg_node_comm[0];
    pos = 0;
    for(j = 0; j < num_nodes; j++)
    {
      if(min>avg_node_comm[j])
      {
            min = avg_node_comm[j];
          pos = j;
```

```
      }
    }
    allocate[0][0]=pos+1;

    for(j = 1; j < num_tasks; j++)
    {
      int count=0,task_dependency[num_tasks][2];
      for(i = 0; i < num_tasks; i++)
      {
            int k=0;
            min2=999999999;
            if(adjacency_matrix[i][j]==1)
            {
                  allo_proc1=(int)allocate[i][0];

                  if(node_comm[allo_proc1-1][k]!=0)
                  {
                        min2 = (node_comm[allo_proc1-
1][k]*task_data[i][j])+exec_pdf[k][j].pdf_mean;
                        pos2=k;
                  }

                for(k = 1; k < num_nodes; k++)
                {
                  x=999999999;
                        if(node_comm[allo_proc1-1][k]!=0)
                        {
                              x=(node_comm[allo_proc1-
1][k]*task_data[i][j])+exec_pdf[k][j].pdf_mean;
                        }

                  if(x<min2)
                  {
                        min2=x;
                        pos2=k;
                  }

                }

                task_dependency[count][0]=min2;
                task_dependency[count][1]=pos2+1;

                  count = count + 1;
            }
      }


      final_task_allo_node = minim(task_dependency,count);
      allocate[j][0]= final_task_allo_node;
    }
}

int minim(int task_dependency[MAX_TASKS][2], int count)
```

```
{
    int i,min,node;
    min = task_dependency[0][0];
    node = task_dependency[0][1];
    for (i = 1; i < count; i++)
    {
      if(task_dependency[i][0] < min)
      {
            min = task_dependency[i][0];
            node = task_dependency[i][1];
      }
    }
    return node;
}


/*********************************************************************/
/*********************************************************************/
/*  write_to_file()                                                  */
/*********************************************************************/
/*********************************************************************/

void write_to_file()
{
    int i;
    char
ipfilename[LINE_LENGTH]="/Users/Uday/Desktop/Files/node_allocation.txt"
;
    FILE *ipf;
    ipf=fopen(ipfilename,"w");

    for(i = 0; i < num_tasks; i++)
    {
      fprintf(ipf,"%d %d %f\n",i+1,(int)allocate[i][0],allocate[i][1]);
    }

    fclose(ipf);
}

/*********************************************************************/
/*********************************************************************/
/*  simulate()                                                       */
/*********************************************************************/
/*********************************************************************/

void simulate()
{
    int status;
    status=system("./Imp1_SimGrid 3nodes.xml");
}

/*********************************************************************/
/*********************************************************************/
```

```
/*  analysis()                                                          */
/*******************************************************************/
/*******************************************************************/

void analysis()
{
    char
ipfilename[LINE_LENGTH]="/Users/Uday/Desktop/Files/execution_times_app_
tm.txt";
    int j;
    float sum=0, sdsum=0, mean, standard_deviation, variance, ci_low,
ci_high, confidence_interval=0.99;

    inp_f = fopen(ipfilename,"r");

    for(j = 0; j < NUM_ITERATION; j++)
    {
      fscanf(inp_f, "%f", &tet[j]);
      sum = sum + tet[j];
    }

    mean = sum / NUM_ITERATION;

    printf(" \n MEAN of TOTAL EXECUTION TIME is %f \n", mean);

    for(j = 0; j < NUM_ITERATION; j++)
    {
      sdsum = sdsum + pow((tet[j] - mean),2) ;
    }

    variance = sdsum/NUM_ITERATION;
    standard_deviation = sqrt(variance);

    ci_low = mean -
(confidence_interval*(standard_deviation/sqrt(NUM_ITERATION)));

    ci_high = mean +
(confidence_interval*(standard_deviation/sqrt(NUM_ITERATION)));

    printf("\n 99 percent Confidence Interval values are\n");
    printf("\n LOW = %f , HIGH = %f\n\n",ci_low,ci_high);
}
```

```
/*****************************************************************
** FILE NAME: Imp1_SimGrid.c
** AUTHOR: UDAY KUMAR B NARAYANAPPA
** DATE:  11/01/2010
** DESCRIPTION: gets the task allocation data, the functional
**              description and schedules tasks on nodes
** OUTPUT: total execution time of the complex application
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include<string.h>
#include "simdag/simdag.h"
#include "xbt/ex.h"
#include "xbt/log.h"
#define LINE_LENGTH 200
#define maxtask 1000
#define maxnode 100

XBT_LOG_NEW_DEFAULT_CATEGORY(sd_test,
                             "Logging specific to this SimDag
example");

typedef struct{
    int task;
    int flag;
}mystruct;

mystruct current_schedule[maxtask]; // root tasks that are yet to be
scheduled
mystruct scheduled_tasks[maxtask];  // tasks that are already scheduled
mystruct completed_tasks[maxtask];  // tasks that completed execution

typedef struct {
    int taskname;
    int wsno;
    float task_exec_time;
}element;
element node_allocation[maxtask];   // node allocation data

int schedule_no, paths_count, max_path_len, tasknum, nodenum,
completed_task_count, scheduled_task_count, task_uday;
char name[10];

const int workstation_number = 1;
const char *platform_file;
const SD_workstation_t *workstations;
double power=50000000;  // power of each workstation
double computation_amount[1];
double communication_amount[1];
double rate = -1.0, total_time=0;
SD_task_t tname[maxtask]; // task names
SD_task_t *changed_tasks;
```

```
SD_workstation_t workstation_list[maxnode]; // nodes in the grid
network

void meta_scheduler_schedule(void);
void update_completed_tasks(void);
int func_desc[maxtask][maxtask];
void tm_read_func_desc(void);
void tm_notify_ms(void );
int check_all_completed(void);
int check_task_scheduled(int taskname);
int md_check(int taskname, int col);

int main(int argc, char **argv)
{
    int i,j;
    double end_time, start_time;

    computation_amount[0] = 1;
    communication_amount[0] = 10;


    char
node_allocation_file[LINE_LENGTH]="/Users/Uday/Desktop/Files/node_alloc
ation.txt";
    char
meta_scheduler_file[LINE_LENGTH]="/Users/Uday/Desktop/Files/meta_schedu
ler_3n.txt";
    char
output_exec_file[LINE_LENGTH]="/Users/Uday/Desktop/Files/execution_time
s_app_tm.txt";

    FILE *naf, *msf, *oef;
    naf=fopen(node_allocation_file,"r");
    msf=fopen(meta_scheduler_file,"r");
    oef=fopen(output_exec_file,"a");

    // read the number of tasks from meta scheduler file
    fscanf(msf, "%d", &tasknum);

    // read the number of nodes from meta scheduler file
    fscanf(msf, "%d", &nodenum);

    // Task Manager reads the fucntional description file
    tm_read_func_desc();

    // initialisation of SimGrid
    SD_init(&argc, argv);

    if (argc < 2) {
      INFO1("Usage: %s platform_file", argv[0]);
      INFO1("example: %s sd_platform.xml", argv[0]);
      exit(1);
    }
```

```
    // creation of the environment
    platform_file = argv[1];
    SD_create_environment(platform_file);
    workstations = SD_workstation_get_list();

    for(i=0;i<nodenum;i++)
      workstation_list[i] = workstations[i];

    // read the node allocation data
    for(i=0;i<tasknum;i++)
    {
      fscanf(naf, "%d", &node_allocation[i].taskname);
      fscanf(naf, "%d", &node_allocation[i].wsno);
      fscanf(naf, "%f", &node_allocation[i].task_exec_time);
    }

    // print the node allocation data
    printf("TASK_NAME   WORKSTATION EXEC_TIME\n");
    for(i=0;i<tasknum;i++)
      printf("%d        %d
%f\n",node_allocation[i].taskname,node_allocation[i].wsno,node_allocati
on[i].task_exec_time);


    /* creation of the tasks */
    for(i=0;i<tasknum;i++)
    {
      sprintf(name,"%d",i+1);
      tname[i] = SD_task_create(name, NULL,
power*node_allocation[i].task_exec_time);
    }


    /* watch points */
    for(i=0;i<tasknum;i++)
    {
      SD_task_watch(tname[i], SD_DONE);
    }


    for(j=1;j<=tasknum;j++)
    {

      // Task Manager notifies Meta Scheduler about the root tasks
      tm_notify_ms();

      // print the root tasks
      printf("\n Current tasks to be executed\n");
      for(i=0;i<tasknum;i++)
            printf("\n %d - %d ",current_schedule[i].task,
current_schedule[i].flag);
```

```
      // Meta Scheduler schedules root task
      meta_scheduler_schedule();

      // note the start time of simulation
      start_time=SD_get_clock();

      // start the simulation
      changed_tasks = SD_simulate(-1.0);

      // note the end time of simulation
      end_time = SD_get_clock();

      // print tasks already scheduled
      printf("\nTasks already scheduled\n");
      for(i=0;i<tasknum;i++)
            printf("\n %d - %d ", scheduled_tasks[i].task,
scheduled_tasks[i].flag);

      // update the completed tasks
      update_completed_tasks();

      //total_time is the total execution time
      total_time = total_time + end_time - start_time;


    }


    printf("\nTotal Execution Time for entire task graph is %f
\n",total_time);

    // write the total execution time to output file
    fprintf(oef,"%f\n", total_time);

    // free the completed tasks
    xbt_free(changed_tasks);

    // Destroy the tasks
    for(i=0;i<tasknum;i++)
    {
            SD_task_destroy(tname[i]);
    }

    // close the files
    fclose(naf);
    fclose(msf);
    fclose(oef);

    // exit the simulation
    SD_exit();

    return 0;
}
```

```
/*********************************************************************/
/*********************************************************************/
/*  meta_scheduler_schedule()                                        */
/*********************************************************************/
/*********************************************************************/

void meta_scheduler_schedule()
{
    int i,j;

    for(i=0;i<tasknum;i++)
    {
      if(current_schedule[i].flag == 1)
      {
            for(j=0;j<nodenum;j++)
            {
                  if(node_allocation[i].wsno==j) workstation_list[0] =
workstations[j];
            }

            SD_task_schedule(tname[i], workstation_number,
workstation_list,
                              computation_amount,
communication_amount, rate);
      }
    }

}



/*********************************************************************/
/*********************************************************************/
/*  tm_read_func_desc()                                              */
/*********************************************************************/
/*********************************************************************/

void tm_read_func_desc()
{
    int j,i;
    char
func_descfile[LINE_LENGTH]="/Users/Uday/Desktop/Files/functional_descri
ption.txt";
    FILE *fd;
    fd = fopen(func_descfile,"r");

    // read the number of paths in fucnitonal description
    fscanf(fd, "%d", &paths_count);

    // read the maximum path length
    fscanf(fd, "%d", &max_path_len);
```

```
    // read the fucntional description - if paths are not of same
length, use zeros to make it even
    for(i=0;i<paths_count;i++)
      for(j=0;j<max_path_len;j++)
      {
      fscanf(fd, "%d", &func_desc[i][j]);
      }

    printf("\n Paths are: -\n");

    for(i=0;i<paths_count;i++)
    {
      for(j=0;j<max_path_len;j++)
      {
            printf("%d ",func_desc[i][j]);
      }
      printf("\n");
    }


}

/*********************************************************************/
/*********************************************************************/
/*  update_completed_tasks()                                         */
/*********************************************************************/
/*********************************************************************/

void update_completed_tasks()
{
    int i;
    double time_elapsed=0;

    for (i = 0; changed_tasks[i] != NULL; i++)
    {

        if(SD_task_get_state(changed_tasks[i]) == SD_DONE)
      {
            time_elapsed=SD_task_get_finish_time(changed_tasks[i]) -
SD_task_get_start_time(changed_tasks[i]);
            printf("\nTask '%s' start time: %f, finish time: %f, Time
Elapsed: %f secs \n",
                     SD_task_get_name(changed_tasks[i]),
                     SD_task_get_start_time(changed_tasks[i]),

SD_task_get_finish_time(changed_tasks[i]),time_elapsed);

            completed_tasks[completed_task_count].task =
atoi(SD_task_get_name(changed_tasks[i]));
            completed_tasks[completed_task_count].flag = 1;
            completed_task_count++;
      }
    }
```

```
}

/*********************************************************************/
/*********************************************************************/
/*  tm_notify_ms()                                                   */
/*********************************************************************/
/*********************************************************************/

void tm_notify_ms()
{
    int i, j, k, all_col_tasks=1,dependency_value;

    // initialize current_schedule structure
    for(i=0;i<tasknum;i++)
    {
      current_schedule[i].task=i+1;
      current_schedule[i].flag=0;
    }

    printf("\n Completed tasks are \n");
    for(i = 0 ; i < completed_task_count; i++)
      printf(" %d - %d\n",          completed_tasks[i].task,
completed_tasks[i].flag);


    // for intial schedule, no need to check dependency for tasks in
first column
    if(task_uday==0)
    {
      dependency_value = 1;

      //update the current_schedule structure
      for(i=0;i<paths_count;i++)
      {

            for(k=0;k<tasknum;k++)
            {
                  if(current_schedule[k].task ==
func_desc[i][schedule_no])
                        current_schedule[k].flag = dependency_value;
            }
      }

      task_uday=1;
    }

    //from second time schedule, find root tasks for scheduling and
update the scheduled_tasks structure
    else
    {

      for(i=0;i<paths_count;i++)
```

```
      {

          //char *p;
            int k,md=0;
            dependency_value = 0;

            // check if a task is not scheduled, if so update the
current_schedule structure and scheduled_tasks structure

            if((check_task_scheduled(func_desc[i][schedule_no])!=1) &&
(func_desc[i][schedule_no]!=0))
            {

                  for(k=0;k<paths_count;k++)
                  {
                        if(i!=k)
                        {
                              if(func_desc[i][schedule_no] ==
func_desc[k][schedule_no])
                              {
                                    md=1;
                              }
                        }
                  }


                  //check the dependency
                  if(md==0)
                  {

                        for(j=0;j<completed_task_count;j++)
                        {

                              if(completed_tasks[j].task ==
func_desc[i][schedule_no-1])
                                    dependency_value = 1;
                        }
                  }

                  else
                  {
                        dependency_value =
md_check(func_desc[i][schedule_no],schedule_no);
                  }


                  for(k=0;k<tasknum;k++)
                  {
                        if(current_schedule[k].task ==
func_desc[i][schedule_no])
                              current_schedule[k].flag =
dependency_value;
                  }
```

```
            }
      }//for


    } //else

    //update schedule_tasks structure by scanning current_schedule
structure
    for(i=0;i<tasknum;i++)
    {
      if(current_schedule[i].flag==1)
      {
            scheduled_tasks[scheduled_task_count].task =
current_schedule[i].task;
            scheduled_tasks[scheduled_task_count].flag = 1;
            scheduled_task_count++;
      }
    }

    //check all tasks in the column is scheduled
    for(i=0;i<paths_count;i++)
    {
      if((func_desc[i][schedule_no])!=0)
      {
        if(check_task_scheduled(func_desc[i][schedule_no])!=1)
        {
            all_col_tasks = 0;
        }
      }
    }

    //increment column if all tasks in the column is scheduled
    if(all_col_tasks == 1)
    {
      schedule_no++;
    }

}

//if a task depends on 2 or more tasks, then check whether that
dependency is satisfied or not
int md_check(int taskname, int col)
{
    int j, k, dependency_value=1,f;

    for(k=0;k<paths_count;k++)
    {
      f = 0;
      if(taskname == func_desc[k][col])
      {
            for(j=0;j<completed_task_count;j++)
            {
```

```
                    if(completed_tasks[j].task == func_desc[k][col-1])
                          f = 1;
              }
              if(f==1)
              {
                    dependency_value=dependency_value && 1;
              }
              else
              {
                    dependency_value=dependency_value && 0;
              }
        }

    }

    return dependency_value;

}


/*********************************************************************/
/*********************************************************************/
/*  check_task_scheduled()                                           */
/*********************************************************************/
/*********************************************************************/

int check_task_scheduled(int taskname)
{
    int i,flag=0;
    for(i=0;i<tasknum;i++)
    {
      if(scheduled_tasks[i].task == taskname)
            flag = 1;
    }
    if(flag == 1)
      return 1;
    else
        return 0;
}
```

Implementation II

```
/*********************************************************************
** FILE NAME: Imp2_Main.c
** AUTHOR: UDAY KUMAR B NARAYANAPPA
** DATE:  11/01/2010
** DESCRIPTION: gets the probability of execution time of each task on
**              on each node, communication costs between nodes, and
**              allocates node to tasks and then generate random
**              numbers and simulates the complex application.
**  OUTPUT: 1000 instances of total execution time and MTET values
*********************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include "simdag/simdag.h"
#include "xbt/ex.h"
#include "xbt/log.h"

#define LINE_LENGTH 200
#define MAX_TASKS 1000  // maximum size of task graph
#define MAX_NODES 100   // maximum size of grid network
#define NUM_ITERATION 1000 // number of simulation cycle

int num_tasks;                // number of tasks limited by MAX_TASKS
int num_nodes;                // number of nodes limited by MAX_NODES
char distribution;          // distribution type
char heu;                     // heuristic type
float exec_times[MAX_NODES][MAX_TASKS]; // execution time of tasks on
different nodes
float allocate[MAX_TASKS][2]; // allocation matrix
int node_comm[MAX_NODES][MAX_NODES]; // nodes communication cost
int adjacency_matrix[MAX_TASKS][MAX_TASKS]; // adjacency matrix of task
graph
int task_data[MAX_TASKS][MAX_TASKS]; // task data dependency
float avg_node_comm[MAX_NODES]; // avg. node communication
float tet[NUM_ITERATION]; // total execution time

FILE *inp_f, *out_f;

typedef struct{
    float pdf_mean;
    float pdf_variance;
} PDF;

PDF exec_pdf[MAX_NODES][MAX_TASKS]; // the matrix for execution times
pdf

void read_input_file(void);
```

```
void calc_exec_times(void);
void seetf(void);
void mdpcc(void);
void change_exectimes(void);
void write_to_file(void);
int minim(int task_dependency[MAX_TASKS][2], int count);
float random_exponential(float mean);
float random_uniform(float begin, float end);
float random_normal(float mean, float std);
void analysis(void);
void simulate(void);

int main()
{
    int i, k;

    // read the meta scheduler file
    read_input_file();

    // user selected heuristic
    if(heu == 's')
      seetf();
    else if(heu == 'm')
        mdpcc();

    // experiment is repeated for NUM_ITERATION iterations
    for(k=1;k<=NUM_ITERATION;k++)
    {

      // generate random variates of selected distribution
      calc_exec_times();

      // update the allocation matrix with generated random variates
      change_exectimes();

      // print the allocation matrix
      printf("\n \n");
      for(i = 0; i < num_tasks; i++)
            printf("\n%d %f\t",(int)allocate[i][0],allocate[i][1]);
      printf("\n \n");

      // write the allocation matrix into node_allocation.txt file
      write_to_file();

      // call the SimGrid program to start the simulation
      simulate();

    }

    // calculate mean of total execution time(MTET) and its 99%
confidence interval values
    analysis();
```

```
    return 1;

}

/*************************************************************************/
/*************************************************************************/
/*  read_input_file()                                                    */
/*************************************************************************/
/*************************************************************************/

void read_input_file()
{
    char
ipfilename[LINE_LENGTH]="/Users/Uday/Desktop/Files/meta_scheduler_3n.tx
t";
    int i,j;

    inp_f = fopen(ipfilename,"r");

    // get distribution type
    printf("\n Choose the type of Distribution.\n Enter 'n' for Normal,
'u' for Uniform, 'e' for Exponential\n");
    scanf("%s",&distribution);

    //get heuristic type
    printf("\n Choose the Heuristic.\n Enter 's' for Shortest Estimated
Execution Time First, 'm' for Minimum Data Packets and Communication
Cost\n");
    scanf("%s",&heu);

    // read the number of tasks
    fscanf(inp_f, "%d", &num_tasks);

    // read the number of nodes
    fscanf(inp_f, "%d", &num_nodes);

    // create the adjacency matrix
    for(i = 0; i < num_tasks; i++)
      for(j = 0; j < num_tasks; j++)
        {
            fscanf(inp_f, "%d", &task_data[i][j]);
            if(task_data[i][j]!=0)
                  adjacency_matrix[i][j]=1;
            else
                  adjacency_matrix[i][j]=0;
      }

    // get the means of execution time
    for(i = 0; i < num_nodes; i++)
      for(j = 0; j < num_tasks; j++)
            fscanf(inp_f, "%f", &exec_pdf[i][j].pdf_mean);

    // get the variances
```

```
    for(i = 0; i < num_nodes; i++)
      for(j = 0; j < num_tasks; j++)
            fscanf(inp_f, "%f", &exec_pdf[i][j].pdf_variance);

    // get the nodes communication cost
    for(i = 0; i < num_nodes; i++)
      for(j = 0; j < num_nodes; j++)
            fscanf(inp_f, "%d", &node_comm[i][j]);


    fclose(inp_f);

}


/*********************************************************************/
/*********************************************************************/
/*  calc_exec_times()                                                */
/*********************************************************************/
/*********************************************************************/

void calc_exec_times()
{
    int i,j;

    // generate exponential distribution random variates
    if(distribution == 'e')
    {
      printf("exponential\n");
      for(i = 0; i < num_nodes; i++)
            for(j = 0; j < num_tasks; j++)

    exec_times[i][j]=random_exponential(exec_pdf[i][j].pdf_mean);
    }

    // generate normal distribution random variates
    else if(distribution == 'n')
    {
      printf("normal\n");
        for(i = 0; i < num_nodes; i++)
            for(j = 0; j < num_tasks; j++)

    exec_times[i][j]=random_normal(exec_pdf[i][j].pdf_mean,exec_pdf[i][
j].pdf_variance);
    }

    // generate uniform distribution random variates
    else if(distribution == 'u')
    {
      printf("uniform\n");
      for(i = 0; i < num_nodes; i++)
            for(j = 0; j < num_tasks; j++)
```

```
    exec_times[i][j]=random_uniform(exec_pdf[i][j].pdf_mean,exec_pdf[i]
[j].pdf_variance);
    }
}

/*********************************************************************/
/*********************************************************************/
/*  random_uniform(mean, variance)                                   */
/*********************************************************************/
/*********************************************************************/

float random_uniform(float begin, float end)
{
    float ran_num;
    float mean, width, diff;

    width = end - begin;
    if(width < 0)
      printf("\nINVALID INPUT MEAN, VARIANCE\n");
    mean = (begin + end) / 2;
    diff = mean - (width / 2);

    ran_num = rand() / (float) RAND_MAX;
    ran_num = ran_num * width;
    ran_num = ran_num + diff;
    ran_num = ran_num + begin;

    return(ran_num);

}


/*********************************************************************/
/*********************************************************************/
/*  random_normal(mean, variance)                                    */
/*********************************************************************/
/*********************************************************************/

float random_normal(float mean, float std)
{

    int i;
    float z, ran_num;

    z = 0;
    for( i = 0; i < 12; i++)
      z = z + (rand() / (float) RAND_MAX);
    z = z - 6;  // N(0,1)

    ran_num = z * std + mean; // x = z* standard deviation + mean

    return(ran_num);
```

```
}

/*********************************************************************/
/*********************************************************************/
/*  random_exponential(mean)                                         */
/*********************************************************************/
/*********************************************************************/

float random_exponential(float mean)
{

    float ran_num;

    ran_num = 0;
    while(ran_num == 0)
        ran_num = rand() / (float) RAND_MAX;

    ran_num = mean * log(ran_num);
    ran_num = - ran_num;

    return(ran_num);

}

/*********************************************************************/
/*********************************************************************/
/*  change_exectimes()                                               */
/*********************************************************************/
/*********************************************************************/

void change_exectimes()
{
    int i,j;
    for(i = 0; i < num_tasks; i++)
    {
      j=(int)allocate[i][0];
      allocate[i][1]=exec_times[j-1][i];
    }
}


/*********************************************************************/
/*********************************************************************/
/*  SEETF heuristic                                                  */
/*********************************************************************/
/*********************************************************************/

void seetf()
{
    int i,j,pos;
    double min;
    for(i = 0; i < num_tasks; i++)
    {
```

```
      min=exec_pdf[0][i].pdf_mean;
      pos=0;
      for(j = 1; j < num_nodes; j++)
      {
            if(exec_pdf[j][i].pdf_mean < min)
            {
                  min=exec_pdf[j][i].pdf_mean;
                  pos=j;
            }
      }
      allocate[i][0]=pos+1;
    }
}


/*********************************************************************/
/*********************************************************************/
/*  MDPCC heuristic                                                  */
/*********************************************************************/
/*********************************************************************/

void mdpcc()
{

    int i,j,pos,pos2,allo_proc1,final_task_allo_node;
    float min,min2,x;

    // calculate average node communication
    for(i = 0; i < num_nodes; i++)
    {
      int sum=0;
      for(j = 0; j < num_nodes; j++)
      {
            sum = sum + node_comm[i][j];
      }
      avg_node_comm[i] = sum / num_nodes;
    }

    // first task is allocated a node which has minimum average node
communication
    min = avg_node_comm[0];
    pos = 0;
    for(j = 0; j < num_nodes; j++)
    {
      if(min>avg_node_comm[j])
      {
            min = avg_node_comm[j];
          pos = j;
      }
    }
    allocate[0][0]=pos+1;

    for(j = 1; j < num_tasks; j++)
```

```
    {
      int count=0,task_dependency[num_tasks][2];
      for(i = 0; i < num_tasks; i++)
      {
            int k=0;
            min2=999999999;
            if(adjacency_matrix[i][j]==1)
            {
                  allo_proc1=(int)allocate[i][0];

                  if(node_comm[allo_proc1-1][k]!=0)
                  {
                        min2 = (node_comm[allo_proc1-
1][k]*task_data[i][j])+exec_pdf[k][j].pdf_mean;
                        pos2=k;
                  }

               for(k = 1; k < num_nodes; k++)
               {
                  x=999999999;
                        if(node_comm[allo_proc1-1][k]!=0)
                        {
                              x=(node_comm[allo_proc1-
1][k]*task_data[i][j])+exec_pdf[k][j].pdf_mean;
                        }

                  if(x<min2)
                  {
                        min2=x;
                        pos2=k;
                  }

               }

               task_dependency[count][0]=min2;
               task_dependency[count][1]=pos2+1;

                  count = count + 1;
            }
      }


      final_task_allo_node = minim(task_dependency,count);
      allocate[j][0]= final_task_allo_node;
    }
}

int minim(int task_dependency[MAX_TASKS][2], int count)
{
    int i,min,node;
    min = task_dependency[0][0];
    node = task_dependency[0][1];
    for (i = 1; i < count; i++)
```

```
    {
      if(task_dependency[i][0] < min)
      {
            min = task_dependency[i][0];
            node = task_dependency[i][1];
      }
    }
    return node;
}


/*********************************************************************/
/*********************************************************************/
/*  write_to_file()                                                  */
/*********************************************************************/
/*********************************************************************/

void write_to_file()
{
    int i;
    char
ipfilename[LINE_LENGTH]="/Users/Uday/Desktop/Files/node_allocation.txt"
;
    FILE *ipf;
    ipf=fopen(ipfilename,"w");

    for(i = 0; i < num_tasks; i++)
    {
      fprintf(ipf,"%d %d %f\n",i+1,(int)allocate[i][0],allocate[i][1]);
    }

    fclose(ipf);
}

/*********************************************************************/
/*********************************************************************/
/*  simulate()                                                       */
/*********************************************************************/
/*********************************************************************/

void simulate()
{
    int status;
    status=system("./Imp2_SimGrid 3nodes.xml");
}

/*********************************************************************/
/*********************************************************************/
/*  analysis()                                                       */
/*********************************************************************/
/*********************************************************************/

void analysis()
```

```
{
    char
ipfilename[LINE_LENGTH]="/Users/Uday/Desktop/Files/execution_times_app.
txt";
    int j;
    float sum=0, sdsum=0, mean, standard_deviation, variance, ci_low,
ci_high, confidence_interval=0.99;

    inp_f = fopen(ipfilename,"r");

    for(j = 0; j < NUM_ITERATION; j++)
    {
      fscanf(inp_f, "%f", &tet[j]);
      sum = sum + tet[j];
    }

    mean = sum / NUM_ITERATION;

    printf(" \n MEAN of TOTAL EXECUTION TIME is %f \n", mean);

    for(j = 0; j < NUM_ITERATION; j++)
    {
      sdsum = sdsum + pow((tet[j] - mean),2) ;
    }

    variance = sdsum/NUM_ITERATION;
    standard_deviation = sqrt(variance);

    ci_low = mean -
(confidence_interval*(standard_deviation/sqrt(NUM_ITERATION)));

    ci_high = mean +
(confidence_interval*(standard_deviation/sqrt(NUM_ITERATION)));

    printf("\n 99 percent Confidence Interval values are\n");
    printf("\n LOW = %f , HIGH = %f\n\n",ci_low,ci_high);
}
```

```
/*****************************************************************
** FILE NAME: Imp2_SimGrid.c
** AUTHOR: UDAY KUMAR B NARAYANAPPA
** DATE:  11/01/2010
** DESCRIPTION: gets the task allocation data, task data dependencies
**              and schedules tasks on nodes
** OUTPUT: total execution time of the complex application
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include<string.h>
#include "simdag/simdag.h"
#include "xbt/ex.h"
#include "xbt/log.h"
#define LINE_LENGTH 200
#define maxtask 1000 // maximum number of tasks
#define maxnode 100 // maximum number of nodes

XBT_LOG_NEW_DEFAULT_CATEGORY(sd_test,
                             "Logging specific to this SimDag
example");

int main(int argc, char **argv)
{

    typedef struct {
      int taskname;
      int wsno;
      float task_exec_time;
    }element;

    int i, j, tasknum, nodenum, adjacency_matrix[maxtask][maxtask];
    element node_allocation[maxtask];
    const char *platform_file;
    const SD_workstation_t *workstations;
    SD_workstation_t workstation_list[maxnode];
    double time_elapsed;
    char name[10];
    SD_task_t tname[maxtask];; //task names
    SD_task_t *changed_tasks;
    const int workstation_number = 1;
    double power=50000000;    // power of each workstation
    double computation_amount[1];
    double communication_amount[1];
    double rate = -1.0, end_time, start_time;


    char
node_allocation_file[LINE_LENGTH]="/Users/Uday/Desktop/Files/node_alloc
ation.txt";
    char
meta_scheduler_file[LINE_LENGTH]="/Users/Uday/Desktop/Files/meta_schedu
```

```
ler_3n.txt";
    char
output_exec_file[LINE_LENGTH]="/Users/Uday/Desktop/Files/execution_time
s_app.txt";

    FILE *naf, *msf, *oef;
    naf=fopen(node_allocation_file,"r");
    msf=fopen(meta_scheduler_file,"r");
    oef=fopen(output_exec_file,"a");


    // initialisation of SimGrid
    SD_init(&argc, argv);

    if (argc < 2) {
      INFO1("Usage: %s platform_file", argv[0]);
      INFO1("example: %s sd_platform.xml", argv[0]);
      exit(1);
    }

    // creation of the environment
    platform_file = argv[1];
    SD_create_environment(platform_file);
    workstations = SD_workstation_get_list();

    for(i=0;i<nodenum;i++)
      workstation_list[i] = workstations[i];

    computation_amount[0] = 1;
    communication_amount[0] = 10;

    // read the number of tasks from meta scheduler file
    fscanf(msf, "%d", &tasknum);

    // read the number of nodes from meta scheduler file
    fscanf(msf, "%d", &nodenum);

    // read the node allocation data
    for(i=0;i<tasknum;i++)
    {
      fscanf(naf, "%d", &node_allocation[i].taskname);
      fscanf(naf, "%d", &node_allocation[i].wsno);
      fscanf(naf, "%f", &node_allocation[i].task_exec_time);
    }

    // print the node allocation data
    printf("TASK_NAME   WORKSTATION EXEC_TIME\n");
    for(i=0;i<tasknum;i++)
      printf("%d        %d
%f\n",node_allocation[i].taskname,node_allocation[i].wsno,node_allocati
on[i].task_exec_time);
```

```
    /* creation of the tasks */
    for(i=0;i<tasknum;i++)
    {
      sprintf(name,"%d",i+1);
      tname[i] = SD_task_create(name, NULL,
power*node_allocation[i].task_exec_time);
    }

    // add task data dependency by reading the task data dependency
matrix from meta scheduler file
    for(i = 0; i < tasknum; i++)
      for(j = 0; j < tasknum; j++)
        {
            fscanf(msf, "%d", &adjacency_matrix[i][j]);
            if(adjacency_matrix[i][j]!= 0)
            {
                        SD_task_dependency_add(NULL, NULL, tname[i],
tname[j]);
            }
      }



    // schedule the tasks
    for(i=0;i<tasknum;i++)
    {
            for(j=0;j<nodenum;j++)
            {
                  if(node_allocation[i].wsno==j) workstation_list[0] =
workstations[j];
            }

            SD_task_schedule(tname[i], workstation_number,
workstation_list,
                                    computation_amount,
communication_amount, rate);
    }

    // note the start time of simulation
    start_time=SD_get_clock();

    // start the simulation
    changed_tasks = SD_simulate(-1.0);

    // note the end time of simulation
    end_time= SD_get_clock();

    //when task completes execution, print the execution time taken
    for (i = 0; changed_tasks[i] != NULL; i++) {
        if(SD_task_get_state(changed_tasks[i]) == SD_DONE){
            time_elapsed=SD_task_get_finish_time(changed_tasks[i]) -
SD_task_get_start_time(changed_tasks[i]);
            printf("\nTask '%s' start time: %f, finish time: %f, Time
```

```
Elapsed: %f secs \n",
                    SD_task_get_name(changed_tasks[i]),
                    SD_task_get_start_time(changed_tasks[i]),

SD_task_get_finish_time(changed_tasks[i]),time_elapsed);
      }

    }

    printf("\nTotal Execution Time for entire task graph is %f
\n",end_time-start_time);

    // write the total execution time in output file
    fprintf(oef,"%f\n", end_time-start_time);


    // close the files
    fclose(msf);
    fclose(naf);
    fclose(oef);

    // exit the simulation
    SD_exit();
    return 0;
}
```

```
/*********************************************************************
** FILE NAME: 3nodes.xml
** AUTHOR: UDAY KUMAR B NARAYANAPPA
** DATE:  11/01/2010
** DESCRIPTION: platform file used by SimGrid that describes the
**              topology of the grid network
*********************************************************************/

<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "simgrid.dtd">
<platform version="2">

  <host id="C1" power="50000000"/>
  <host id="C2" power="50000000"/>
  <host id="C3" power="50000000"/>

  <link id="6" bandwidth="125000000" latency="0.000100"/>
  <link id="2" bandwidth="125000000" latency="0.000100"/>
  <link id="14" bandwidth="1250000000" latency="0.001000"/>
  <link id="8" bandwidth="125000000" latency="0.000100"/>
  <link id="1" bandwidth="125000000" latency="0.000100"/>
  <link id="4" bandwidth="125000000" latency="0.000100"/>
  <link id="0" bandwidth="125000000" latency="0.000100"/>
  <link id="13" bandwidth="1250000000" latency="0.000100"/>

  <route src="C1" dst="C2"><link:ctn id="6"/><link:ctn
id="14"/><link:ctn id="2"/></route>
  <route src="C2" dst="C1"><link:ctn id="2"/><link:ctn
id="14"/><link:ctn id="6"/></route>
  <route src="C1" dst="C3"><link:ctn id="4"/><link:ctn
id="1"/><link:ctn id="0"/></route>
   <route src="C3" dst="C1"><link:ctn id="0"/><link:ctn
id="1"/><link:ctn id="4"/></route>
  <route src="C2" dst="C3"><link:ctn id="1"/><link:ctn
id="8"/><link:ctn id="13"/></route>
  <route src="C3" dst="C2"><link:ctn id="13"/><link:ctn
id="8"/><link:ctn id="1"/></route>

</platform>
```