RESOURCE AWARE LOAD DISTRIBUTION STRATEGIES FOR SCHEDULING DIVISIBLE LOADS ON LARGE-SCALE DATA INTENSIVE COMPUTATIONAL GRID SYSTEMS

SIVAKUMAR VISWANATHAN

(*M.Sc., National University of Singapore*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2008

Acknowledgments

It is a pleasure to thank the people who contributed in some way to this thesis.

First, I would like to express my sincere gratitude to my supervisor, Assoc. Prof. Bharadwaj Veeravalli. He inspired me with his enthusiasm and helped me to understand the nuances of divisible load scheduling. Throughout my candidature, he provided constant encouragement, sound advices, and lots of good ideas to pursue on. At times, when I felt lost in the woods he guided me to read the stars in the sky and explore my way. I would probably have been lost without him and his style of guidance.

I am grateful to Prof. Thomas G. Robertazzi of Stony Brook University and Dr. Dantong Yu of Brookhaven National Laboratory (BNL) for their valuable guidance and comments on my research work.

I would like to express my gratitude to my employers Institute for Infocomm Research (I^2R) for supporting me during this part-time study. I am grateful to Dr. Michael Li Ming, who convinced me to pursue Ph.D. degree, Prof. Wong Wai Choong Lawrence, Prof. Lye Kin Mun, Mr. Cheah Kok Beng, and Mr.

Ashok Kumar Marath for their continuous encouragement and support during this pursuit.

I would like to thank Mr. T.V. Karthikeyan, my first project manager at Indira Gandhi Centre for Atomic Research (IGCAR), India, who initiated me to the world of designing scheduling strategies.

I wish to thank Mr. Jean-Luc Lebrun who helped to horn my technical writing skills.

I am indebted to my fellow student colleagues Dr. Zeng Zeng, Mr. Jia Jingxi, Mr. Steven He, Mr. Liu Yanhong and Mr. Goh Lee Kee for the stimulating discussions and also their help in working with LaTeX.

I would like to thank Ms. Suzanne Koh and Ms. Indrani Kaliyaperumal, secretaries in Department of Electrical and Computer Engineering, NUS for assisting me in the adminstrative matters during my candidature.

I wish to thank my brother, sisters, in-laws and their families for providing me an environment of love and understanding.

Finally, I would like to thank my parents, Viswanathan and Prema, for their support, teachings, love, and encouragement all through these years; my wife Lalitha and kids Bavadharini and Varun, for their understanding, support, patience, and sacrifices, which gave me the width required to make this possible. It is to them, I dedicate this thesis.

Contents

Summary

Complex scientific problems, as in the large volume of data that are being generated in the high energy nuclear physics experiments, bio-informatics, astronomical computations etc, demand new strategies for how the data is to be collected, shared, transferred and analyzed. Also, the technologies are continuously improving and over the years, the computing power, data storage and networking technologies are seen to grow exponentially. Grid computing paradigm evolved because of these expanding collaborations, data analysis requirements and increasing computational and networking capabilities. Grid is generally viewed as a repository of resources that can be availed by careful scheduling.

In this thesis, we design and analyze several polynomial-time complex, resource aware scheduling strategies for handling computationally intensive arbitrarily divisible loads in a computational Grid system comprising of clusters of computing systems interconnected by high speed links. Computational Grid systems require a hierarchy of scheduling strategies, since the communication delay is considered to be insignificant within clusters while it is significant across clusters because of

their geographical distribution. The design of our proposed strategies adopt the divisible load paradigm, referred to as divisible load theory (DLT), which is shown to be efficient in handling large volume arbitrarily divisible loads.

We propose several strategies, namely

- Dynamic IBS algorithms
- Adaptive IBS algorithm, and
- Resource aware dynamic incremental scheduling algorithm (RADIS) with non-interleaved, earliest deadline first and progressive interleaved scheduling strategies

for distributing the loads within clusters, involving multiple sources (with loads to be processed) and sinks (the processing nodes). We assume a multi-port communication model and devise "pull-based" (the sinks request load from the sources) strategies. All our strategies utilize buffer reclamation approach to schedule the processing of loads. We consider real-life scenario wherein there are finite buffer constraints at the sinks and the loads have deadlines. We propose efficient scheduling strategies with admission control policy that ensures that the admitted loads are processed satisfying their deadline requirements. We demonstrate detailed workings of the proposed algorithms via a simulation study using real-life parameters obtained from a major physics experiment.

We also propose

- Resource aware sequential load distribution strategy (RASLD) and
- Resource aware parallel load distribution strategy (RAPLD)

for scheduling across heterogeneous cluster nodes interconnected by heterogeneous links in an arbitrary manner, assuming a uni-port communication model. We apply various spanning tree construction strategies such as

- Minimum spanning tree (MST)
- Shortest path spanning tree (SPT)
- Fewest hops spanning tree (FHT)
- Robust spanning tree (RST), and
- Minimum network equivalence spanning tree (EST)

with our distribution strategies following the optimal sequencing theorem presented in the literature. We evaluate the performance of the proposed strategies over a wide range of arbitrary dense graphs with varying connectivity (link) and node densities. We also study the effect of network scalability and recommend distribution strategies that provide a better trade-off between complexity and time performance under various scenarios.

All the proposed scheduling strategies are scalable, relevant in real-life situations and are shown to be useful under different scenarios.

List of Tables

List of Figures

List of Symbols

G	An arbitrary topology graph network.
C	The master (coordinator) nodes in a graph network.
E	The communication links in a graph network.

Scheduling within Clusters :

$\alpha_{i,j}$	Amount of load sink K_j shall request from source S_i in an iteration.
α_j	Fraction of the total load L that sink K_j shall consider in an iteration.
β	Acceptance ratio, defined as the ratio of number of loads accepted to the number of loads arrived at a system.
$B_j^{(q)}$ $(\hat{B}_j^{(q)})$	Available (Estimated) buffer space in sink K_j in the q^{th} iteration.
$B_{j,t}$	Available buffer space in K_j at time t.
$\hat{B}_j^{(t)}$	Time averaged buffer space availability at sink K_j, estimated based on historical data.

Continued on Next Page...

χ	The ratio of the average buffer utilization in the time-varying buffer scenario (ζ_{TVB}) to the average buffer utilization in the time-invariant buffer scenario (ζ_{TIB}).
η	The ratio of acceptance ratio in the time-varying buffer scenario (β_{TVB}) to the acceptance ratio in the time-invariant buffer scenario (β_{TIB}).
γ	Throughput of the system, defined as the ratio of number of loads processed to the number of loads accepted in a system (at the end of the simulation period).
λ	Load arrival rate, i.e. number of load arrivals per second.
L_i	Load at source S_i such that the total load in the system, $L = \sum_{i=1}^{N} L_i$.
L_j	Difference between the estimated and actual amount of load processed at sink K_j in an iteration.
M	Total number of sinks in the system, with each sink denoted by $K_j,\ j = 1, ...M$.
N	Total number of sources in the system, with each source denoted by $S_i,\ i = 1, ...N$.
Φ	The number of loads accepted in the time-varying buffer scenario.
p	The confidence level of the buffer estimator, i.e. the probability that the estimated buffer size will be available at a sink at the next iteration.
P_{all} (P_{now})	Set of sinks (with buffer space available for processing in an iteration) in the system.
q	Iteration index.
s	Window size used for estimating the buffer availability based on historical data.

Continued on Next Page...

t_{next} (t_{prev})	Time at which the buffer space at K_j changes again (changed earlier).
T	Current time in the system.
$\hat{T}$	Estimated processing time for the admitted loads in the system.
$T^{(q)}$	Time taken to process the loads in the q^{th} iteration.
T_{cp} (T_{cm})	Computing (Communication) intensity constant. Time taken to process (communicate) a unit load by a standard node (link).
T_{d_i}	Deadline requirement of the source S_i.
T_{opt}	Optimal computation time for the loads in the system.
T_{ul}	Time required to process a unit load.
w_j	Inverse of the computing speed of the sink K_j.
X_{new}	Set of sources that arrive at the system when the system is idle or busy processing for some sources.
X_{now} (X_{later})	Set of sources that are being processed in an iteration (shall be processed in a later iteration).
Y	Fraction of the load L that should be taken into consideration in an iteration of installment, where $Y \leq 1$.
$z_{i,j}$	Inverse of the speed of the link $l_{i,j}$ between node S_i and K_j.
ζ	Average buffer utilization in an iteration at a sink node.
Z_{now} (Z_{later})	Set of sources that are being considered (shall be processed later) during the admissibility testing.

Continued on Next Page...

Scheduling across Clusters :

α	This is defined as a N-tuple and refers to a load distribution, i.e., $\alpha = (\alpha_0, \alpha_{0,1}, ..., \alpha_{0,m_0}, ..., \alpha_{x,i}, ..., \alpha_{g,1}, \alpha_{g,2}, ..., \alpha_{g,m_g})$, where $\alpha_{x,i}$ is the fraction of load assigned to $C_{x,i}$ such that $0 \leq \alpha_{x,i} \leq 1$ and sum of all the above load fractions amounts to the total load to be processed. Note that $m_0, ..., m_g$ reflect the degree of connectivity for each sub-tree.
$\alpha_{x,eq(i)}$	Load fraction given to the equivalent node $C_{x,eq(i)}$ for processing. Note that for $C_{eq(0)}$ and its equivalent network $\Sigma(0, m+1)$, we denote this value as $\alpha_{eq(0)} = 1$.
$\alpha_{x,i}$	The optimal load fraction assigned to node $C_{x,i}$ in $\Sigma(x, i, m+1)$. Similarly, for child node $C_{i,k}$, we denote this value as $\alpha_{i,k}$.
C_s	Node s in the given graph G.
$C_{x,eq(i)}$	An equivalent node of the single-level tree network $\Sigma(x, i, m+1)$. That is, we replace the sub-tree rooted at node i with its equivalent node. Note that, node x now becomes the parent of both node i as well as this equivalent node. We denote the respective speed parameter (inverse of the speed) of this equivalent node as $w_{x,eq(i)}$. Similarly, the equivalent node of $\Sigma(0, m+1)$ is denoted as $C_{eq(0)}$.
$C_{x,i}$	This denotes node i in a spanning tree whose parent is node x. For the root node of a spanning tree, we simply denote it as C_0.
δ	The normalized optimal processing time, defined as the ratio of the optimal processing time ($T^*(\alpha^*)$) for RAPLD and RASLD strategies.
ε	The network eccentricity, defined as the distance in number of hops from the root node to the farthest leaf node in a spanning tree.
l_{C_s,C_t}	Communication link connecting nodes s and t in the graph G.

Continued on Next Page...

$l_{C_{x,i},C_{y,j}}$	Communication link connecting processors $C_{x,i}$ and $C_{y,j}$ in a spanning tree.
$l_{x,eq(i)}$	An equivalent link which is equivalent to the communication capability of a set of links in $\Sigma(x,i,m+1)$. The respective speed parameter of this equivalent link is denoted by $z_{x,eq(i)}$.
L	The total amount of load originating at a node for processing.
$L_{x,i}$	This is defined as the total amount of load assigned to $C_{x,i}$ in a spanning tree.
P_{Link}	The degree of connectivity in a network or the link density.
$\Sigma(x,i,m+1)$	This is a single-level tree network (sub-tree) defined in a spanning tree, consisting of m child nodes $C_{i,1},...,C_{i,k},...,C_{i,m}$, with root node $C_{x,i}$. Further, since every child node has the same parent in this sub-tree, we can conveniently denote the communication link $l_{C_{x,i},C_{i,k}}, k \in (1,...,m)$, connecting $C_{x,i}$ with $C_{i,k}$ simply as $l_{i,k}$. Note that for the single-level tree with root node C_0, we denote it as $\Sigma(0,m+1)$.
$T(\alpha)$	The total processing time of the entire load under the distribution α. Note that $T(\alpha) = \max\{T_{x,i}(\alpha)\}$ where the maximization is over all the nodes in the network.
$T(\Sigma(x,i,m+1))$	This is defined as the optimal processing time of the assigned load fraction $\alpha_{x,eq(i)}$ to $\Sigma(x,i,m+1)$. Note that for $\Sigma(0,m+1)$, we denote the optimal processing time as $T(\Sigma(0,m+1))$, which is indeed the processing time of the entire load L.
$T^*(\alpha^*)$	The optimal processing time of a load, which is the minimum processing time to finish the processing of the entire load, using an optimal load distribution α^*.
$T_{x,i}(\alpha)$	The time instant by which processor $C_{x,i}$ stops its computation under the distribution α.

Continued on Next Page...

$w_{x,i}$	A constant that is inversely proportional to the speed of node $C_{x,i}$. Note that, $w_{i,k}$ is the inverse of the speed of node $C_{i,k}$ in $\Sigma(x, i, m+1)$.
$z_{x,i}$	A constant that is inversely proportional to the speed of link $l_{x,i}$. Note that, $z_{i,k}$ is the inverse of the speed of link $l_{i,k}$ in $\Sigma(x, i, m+1)$.

Chapter 1

Introduction

Complex scientific problems rely heavily on the computation and data analysis capabilities offered by the technologies. Even though the computing power, data storage, and communication technologies continue to improve and grow exponentially, computational resources are failing to keep up with the demands from the scientific community. Over the years, the speed of networks, storage capacity, and computing power are seen to double in about 9, 12, and 18 months, respectively [1]. Here, it is pertinent to note that the network speeds quadruple while the computing power doubles in about the same period. To exploit this bandwidth bounty, new ways of collaborative working that are communication intensive, such as pooling computational resources, streaming large amounts of data between instruments and computing systems, and networking sensors and computing resources are essential. Thus, the expanding collaborations and intensive data analysis coupled with increasing computational and networking capabilities stimulated a new era

of service oriented computing, called "Grid computing" [2].

The major characteristics of Grid computing environments are the large-scale coordinated resource sharing, innovative applications, and high-performance computations. Grid computing enables flexible, secure, coordinated resource sharing among dynamic collection of individuals, institutions, and resources. It creates middleware and standards to function between computers and networks to allow full resource sharing among individuals, research institutes, and organizations and to dynamically allocate the idle computing capability to the needed users at remote sites. Generally, resource sharing is conditional: owners make resources available, subject to constraints on when, where, and what can be done with them.

In Grid environments, authentication, authorization, resource discovery, and resource access/scheduling are some of the key challenges. There are ongoing research and development efforts focusing on designing protocols, services, and tools to address the challenges in building scalable virtual organizations for the Grid. These include security solutions aiding credential and policy management for computations spanning across institutions; query mechanisms for sharing information on resources, supported services etc; protocols for secure remote access of resources; and data management services enabling data transfer between storage systems and applications [3].

New Grid infrastructures are being designed and deployed and the middleware is being constantly improved. Grids are being deployed for providing various types of services, such as

- *computational services*: providing secure services for task execution on distributed computational resources [4, 5]
- *data services*: providing access to and management of distributed data [6, 7]
- *application services*: providing transparent access to remote software libraries and utilities [8]
- *information services*: enabling extraction and presentation of data utilizing all the above mentioned services, and
- *knowledge services*: supporting acquiring, storing, retrieving, publishing, and maintaining knowledge.

With the advent of groups with different requirements and objectives into the Grid community, there are research activities focusing on orchestrating workflows in a service-based environment enabling dispatching jobs with assurances on work completion time, performance, cost etc that are negotiated as part of some Service Level Agreements [9]. These address the query *"How to best schedule a given job onto the available resources in a Grid, given that each job has an agreed set of constraints, so as to meet as many constraints as possible?"*

Scheduling in Grid environment is a significant problem in fairly allocating the available resources. Quality of service constraints allow one to submit jobs/tasks with reliable guarantees that they will be processed by certain times. This is a critical function for applications involving real time deadlines (time critical applications), mission critical computing and also lays a foundation for market based

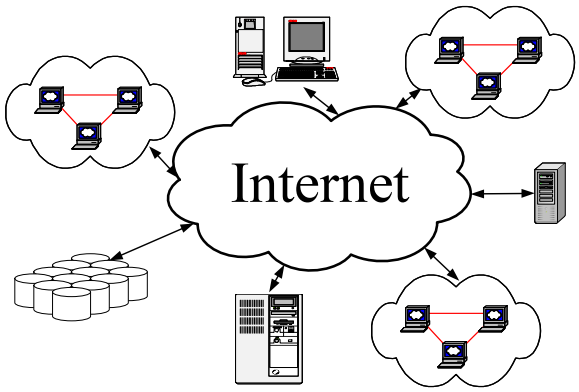

Figure 1.1: Grid infrastructure.

meta-computing. Grid systems operate in dynamic environments and are subject to various unforeseen and unplanned events that can happen at short notice. Such events include sudden failure of computing resources, arrival of new jobs, processing time variations of jobs, resource availabilities etc. The performance of a schedule is very sensitive to these disturbances, and hence it is difficult to execute a predictive schedule generated in advance. These real-time events not only interrupt system operation but also upset the schedules that were previously established. Consequently, the resulting schedule may neither be feasible nor optimal anymore. Recently, memory constrained problem formulation for Grid systems are being considered. Ming and Xian-He [10] studied memory conscious task scheduling for Grid systems. Korkhov et al [11] have proposed a hybrid resource management approach for efficient parallel distributed computing on the Grid, operating on both application and system levels. Kim and Weissman [12] have presented a genetic algorithm approach for decomposable data processing on large scale data Grids. Ruchir et al [13] have proposed job migration algorithms that consider job

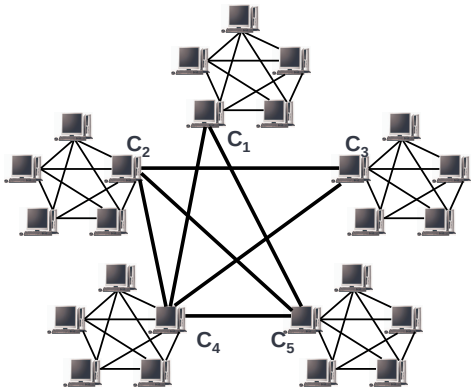

Figure 1.2: A computational Grid system.

transfer cost, resource and network heterogeneity, for load balancing in large and small scale heterogeneous Grid environments.

1.1 Computational Grid Systems

A generic Grid infrastructure comprises of network of supercomputers and/or clusters of computers having different storage, computing and communication capabilities that are inter-connected as shown in Fig. 1.1. The computational Grid systems (CGS) are constructed by using clusters or traditional parallel systems as their nodes as shown in Fig. 1.2. For example,

- the World-Wide Grid, being used for evaluating the Gridbus technologies and applications [14], has many cluster nodes that are located far apart (AIST-Japan, N*Grid Korea, University of Melbourne, and NRC Canada).

- the Dutch Distributed Advanced School for Computing and Imaging (ASCI)

Supercomputer 2 (DAS-2) [15], a Grid infrastructure in the Netherlands located at five Dutch Universities (Vrije Universiteit, University of Amsterdam, Delft University of Technology, Leiden University, and University of Utrecht), built out of clusters of workstations interconnected by Myrinet (a multi-Gigabit LAN used for local communication) and SurfNet (an Internet backbone for wide-area communication).

- the NSF TeraGrid [5] in the United States of America (USA).

1.2 Divisible Load Scheduling

Divisible loads are a class of loads that require homogeneous processing and can be partitioned into arbitrary smaller fractions. These load portions, that bear no dependence relationships among themselves, can then be assigned to individual nodes for processing. Research since 1988 has established that optimal allocation/scheduling of divisible load to nodes and links can be solved through the use of a very tractable linear model formulation, referred to as Divisible Load Theory (DLT). DLT paradigm is proven to be a very useful tool for handling large scale arbitrarily partitionable loads in networked computing environments [16].

DLT can model a wide variety of approaches. For instance, one can distribute the load either sequentially or concurrently. Under sequential load distribution, in most of the literature to date [16–20], the policy used is that a node will distribute load to one of its children at a time. This results in saturating speedup as network

size is increased. One could improve performance by distributing load from a node to children in periodic installments but performance still saturates as the number of installments is increased as shown in [21]. A superior performance results, if load is distributed concurrently. That is, a node distributes load simultaneously to all of its children. Kim [22] has proposed a mathematical model in which simultaneous communication to several nodes is carried out. Juim et al [23] have shown that such concurrent load distribution is scalable for a single level tree when the number of children nodes increases (i.e. linear growth in speedup as the number of children nodes increases).

Other scheduling features that can be modeled are store and forward and virtual cut through switching and the presence or absence of front end processors. Front end processors allow a node to both communicate and compute simultaneously by assuming communication duties. There exists literature of some sixty journal papers on DLT. In addition to the monograph [16], two introductory up-to-date surveys have been published recently [24, 25]. The DLT theory has been proven to be remarkably flexible in the sense that the model allows analytical tractability to derive a rich set of results regarding several important properties of the proposed strategies and to analyze their performance. Agrawal and Jagadish [26] have presented a study on optimal solutions for scheduling "large-grained" computations on loosely coupled processor systems focusing on single-level tree architecture whereas Cheng and Robertazzi [27] considered bus network systems. Real-time optimization of distributed loads originating at various sites of a bus network has also been

studied by Haddad [28]. Marchal et al [29] have considered scheduling divisible loads for generic large scale platforms. In a recent paper Yao and Bharadwaj [30] have proposed strategies for scheduling divisible loads on arbitrary graph networks. Lin et al [31] have studied on providing performance guarantees to divisible load applications in a cluster environment. Another study that may be useful in cluster systems context is by Ghose et al [32] where in time-varying speeds of links and processors in the network are considered in the modeling to evolve an adaptive load distribution strategy.

Scheduling loads under time-varying processor and link speeds have been studied in [33]. An Incremental Balancing Strategy (IBS) has been proposed in [34] for systems with buffer constraints at processing nodes. The IBS algorithm produces a minimum time solution given pre-specified buffer constraints and it also exhibits finite convergence. However, it does not consider scheduling under dynamic environments and buffer capacity variations at processing nodes. Issues such as processor release times coupled with buffer capacity constraints are studied in [35]. In [36] Ghose et al have used a completely novel approach to estimate the speeds of the processors in the network. This study is particularly useful when processor speeds are not known a priori. The solution time (time at which the processed loads/solution is made known at the originator) is discussed in [37]. A completely different objective of minimizing the monetary cost of processing divisible loads is addressed in [38]. In [39] Beaumont et al have discussed some open ended problems and issues pertaining to divisible load scheduling.

DLT has been applied to many real-life applications, including large-scale matrix-vector products [40, 41], large-scale database search problems [42], database application [43, 44], parallel video encoding [45], image processing [46, 47], biological computations [48], optimal pricing study [49], scheduling under system buffer constraints [50], etc. The usefulness of DLT has also been exemplified in the article [24].

DLT paradigm is rich in features, such as, ease of computation, a schematic language, equivalent network element modeling, results for infinite sized networks and numerous applications. This linear model formulation usually produces optimal solutions through linear equation solution or, in simpler models, through recursive algebra. Optimality here involving solution time and speedup is defined in the context of a specific scheduling policy and interconnection topology. The model can take into account heterogeneous node and communication link speeds as well as relative computation and communication intensity. The linear theory formulation opens up striking modeling possibilities for systems incorporating computation and communication issues, as in parallel, distributed and Grid computing.

1.3 Scheduling Divisible Loads on Computational Grids

Computational Grid systems are built on high-speed networks for remote resource usage and thus are well suited for processing large volume arbitrarily divisible data

like those being generated in the high energy and nuclear physics experiments [51], bio-informatics [52], astronomical computations [53], weather prediction etc. The unprecedented volume of data being generated in these applications demand new strategies for how the data is to be collected, shared, transferred and analyzed. For example, the Solenoidal Tracker at RHIC (STAR) experiment at Brookhaven National Laboratories (BNL) is collecting data at the rate of over a Tera-Bytes/day. After the Relativistic Heavy-Ion Collider (RHIC) experiments at BNL came on-line in 1999, STAR began data taking and concurrent data analysis that will last about ten years. STAR performs data acquisition and analyzes over approximately 250 tera-bytes of raw data, 1 peta-bytes of derived and reconstructed data per year. Details on data acquisition and hardware of STAR can be found in [51]. The volume of data is expected to increase by a factor of 10 in the next five years. The STAR collaboration is a large international collaboration of about 400 high energy and nuclear physicists located at 40 institutions in the USA, France, Russia, Germany, Israel, Poland, and so on. These experiments require effective analysis of large amounts of arbitrarily divisible data by widely distributed researchers who must work closely together.

1.4 Our Contributions

The large number and diverse nature of the computing resources and their users in CGS pose a significant challenge to efficiently schedule the loads and utilize

the resources. The motivation for our work stems from the challenges in managing and utilizing computing resources in Grids as efficiently as possible. To-date there has been little or no work on designing resource aware dynamic strategies for scheduling large volume computationally intensive divisible loads with deadline requirements (time critical loads) in a computational Grid environment. In a typical CGS, nodes with in Clusters are co-located and connected by high speed local networks while the Cluster themselves are geographically distributed and are interconnected through wide area networks. Hence, while scheduling large volume computationally intensive arbitrarily divisible loads on the CGS, the communication delay could be ignored while scheduling within Clusters, and it needs to be considered while scheduling across Clusters. Thus, scheduling divisible loads in CGS require multi-level or hierarchy of scheduling strategies.

The main emphasis or the scope of this thesis lies in designing efficient strategies for scheduling large volume computationally intensive divisible loads on CGS and analyzing their performance. We assume the communication delay between the nodes in the system to be contributed by the load transmission time, which is proportional to the size of the load, ignoring the constant propagation delays and the stochastic queuing delays. We also assume a multi-port communication model for scheduling with in clusters (since communication delay is negligible) and design strategies taking into account the influence of heterogeneity in processing capabilities, buffer size variations at the nodes and dynamic arrival of time critical as well as non-critical loads. We employ both interleaving and non-interleaving

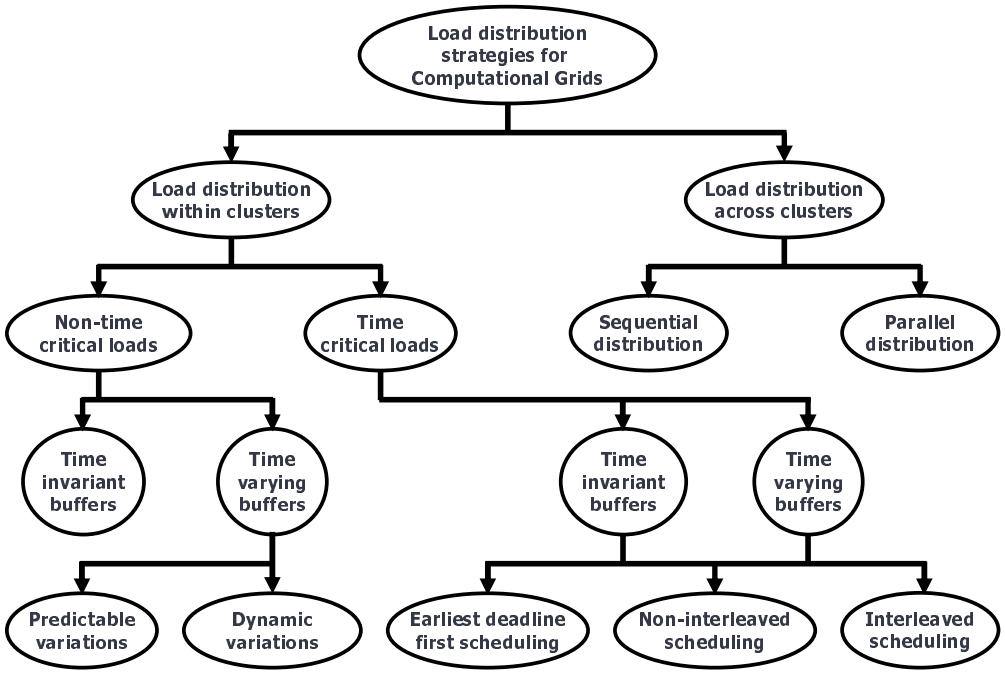

Figure 1.3: Scope of the thesis.

multi-installment strategies to process tasks (jobs) that are admitted into a cluster system, discuss their usefulness and derive important conditions based on which admission control shall be carried out. As communication delays dominate across clusters, we consider them and propose several distribution strategies for inter cluster scheduling assuming a uni-port communication model and quantify their performance. Resource reclaiming strategies are utilized in the design of all our proposed algorithms. In summary, as illustrated in the Fig. 1.3 we propose

- Dynamic iterative strategies for scheduling several non-time critical divisible (partitionable) loads within clusters where there are finite buffer capacity constraints at the processing nodes.

- Resource aware iterative strategies for scheduling several deadline driven

loads within clusters, while adapting to the finite buffer capacity constraints at the processing nodes.

- Load distribution strategies for best scheduling divisible loads on interconnected clusters which forms the backbone network of CGS.

Detailed analysis of the proposed algorithms and their performance are demonstrated using simulation studies with real-life parameters derived from high energy nuclear physics experiments discussed in [51]. The analytical flexibility offered by Divisible Load Theory (DLT) is thoroughly exploited to design resource conscious algorithms that make best use of the available resources.

Since, this study is one of its first kind to address all the above mentioned issues collectively, we propose suite of strategies and analyze their performance by simulation studies. Our systematic design clearly elicits the advantages offered by our strategies. Experimenting on actual Grids is beyond the scope of this thesis and is a challenge in itself.

This thesis is organized as follows: The scheduling problem in CGS is formalized in Chapter 2. The load distribution strategies that are utilized in our scheduling algorithms are described in Chapter 3. Strategies for scheduling non-time critical and time critical loads within a cluster environment are presented in Chapters 4 and 5 respectively. Strategies for scheduling across clusters are explored in Chapter 6 and the conclusions and possible future extensions are in Chapter 7.

Chapter 2

System Modeling and Problem Formulation

In this chapter, we shall describe our system model; introduce the terminology, definition, and notations that are used throughout this thesis.

A computational Grid system (CGS) to be considered here comprises of clusters of computing systems interconnected to form a Grid as shown in Fig. 1.2. We consider the problem of scheduling large volume loads (divisible loads) in such a Grid infrastructure assuming all nodes have front ends. We envisage the cluster system as a *cluster node* comprising a set of computing nodes. Communication delay is assumed to be negligible within a cluster node while it is considered for inter-cluster communications. For network locality, nodes form clusters and each cluster provides a master node, denoted as 'C_s' in Fig. 1.2. All the master nodes

serve as the focal point for their cluster and form the backbone network for inter-cluster communication.

2.1 Scheduling within Cluster Systems

The underlying computing system within a cluster comprising of N control processors, referred to as *sources*, that have load to be processed and M computing elements, referred to as *sinks*, for processing loads, can be modeled as a fully connected bi-partite graph (as in Fig. 2.1): a set of graph vertices could be decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent, while any pair of two graph vertices from these two sets is adjacent. This represents the fact that each source can schedule its load on all the sinks.

All the nodes in the system, in addition to participating in processing the divisible loads from other nodes, also have local tasks to handle. The local tasks needs be processed at the respective nodes. In some systems, the nodes have dedicated buffer spaces for processing divisible loads from other nodes. Such systems are termed as *Systems with time-invariant buffer space availabilities*. Where as in some systems, the nodes share the buffer spaces for processing both local tasks and the divisible loads from other nodes. In such systems, if the local task arrivals and their memory requirements are known a priori, they are termed as *Systems with predictable buffer space availabilities*. If the local task arrivals and their memory requirements vary, the buffer availability at a node also varies over time. Such

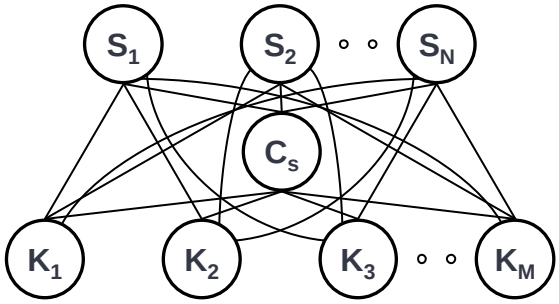

Figure 2.1: Abstract view of a cluster comprising sources & sinks with a coordinator node (C_s) in a Grid system.

systems are termed as *Systems with time-varying buffer space availabilities.*

In real-life situations, one of the practical constraints is in satisfying the deadline requirements of the loads (arriving in real-time from multiple source nodes) to be processed while taking into account the availability of the buffer (memory) resources at the sink nodes, since, the memory available at the processing nodes to store the received load and process them is limited. We consider these combined influences in our proposed algorithms for scheduling with in a cluster. We employ "pull-based" approach in the design of our scheduling strategy wherein the sinks schedule the competing sources depending on the availability of the resources for processing with in a cluster.

The problem that we address shall be formally defined as follows: We consider a cluster node in a Grid system comprising N source nodes denoted as $S_1, S_2, ..., S_N$ and M sink nodes denoted as $K_1, K_2, ..., K_M$. Each source S_i has a load L_i to be processed. In our model, all the nodes in the clusters are assumed to have front-ends. This means that all the nodes can compute and communicate with other

nodes simultaneously. A master node is assumed to coordinate the activities within a cluster. The master node estimates the load distribution and does admission control for the sources. We refer to this master node simply as a *coordinator node* (C_s), and without loss of generality, we assume that any node within a cluster can be elected as the coordinator node based on leader election algorithms [54].

As shown in Fig. 2.1, there are direct links (may be virtual) from all source and sink nodes with in a cluster to C_s. We adopt a simultaneous load distribution model proposed in [55] in which all sources (sinks) can send (receive) load fractions to all the sinks (from all the sources) simultaneously. Also, following Kim's model [22], we assume that the communication time delay is insignificant compared to the time taken for computing, owing to high speed links within clusters, so that no sink starves for load and that all sinks could start computing as they receive the loads from the sources.

The objective here is to schedule and process the loads among M sink nodes, rendering finite buffer capacities, such that their *processing time*, defined as the time instant when all the M sinks complete processing the loads, is a minimum. As with the real-life situation, we consider the availability of buffer space as a time-varying quantity in our formulation and propose multi-installment based scheduling strategies. Also, our objective is to minimize the scheduling related communication overheads in the system. At the start of every iteration, the coordinator node obtains the information about the available memory capacities and computing speeds from the sinks, and the size and deadline requirements of the loads from

the sources. The coordinator node then computes the parameters required by the sinks for scheduling and broadcasts them to all of the sinks. The sink nodes determine the amount of load fractions to be received from the source nodes based on the scheduling parameters received from the coordinator node. The sources, upon receiving the requests from the sinks shall send their load to all sinks concurrently. This process is repeated by the coordinator, sink and source nodes in the system until all the entire loads at the source nodes are processed. Thus, all the proposed schemes for scheduling within clusters in this thesis are distributed strategies and the loads get processed in multiple installments.

In Chapter 3, we describe the load distribution strategy for this multi-source multi-sink environment. In Chapter 4, we propose and analyze Dynamic and Adaptive IBS algorithms, for non-time critical loads with finite buffer constraints at the processing nodes. These algorithms are a generalization of the Modified IBS algorithm [56], tuned to consider dynamic arrival of loads. Then, in Chapter 5, we extend it to design Resource Aware Dynamic Incremental Scheduling (RADIS) strategies that consider loads with deadlines. Admissibility criteria to handle loads with deadlines are also proposed. Detailed analysis of the proposed algorithms and their performance are demonstrated using a simulation study with real-life parameters derived from high energy nuclear physics experiments discussed in [51].

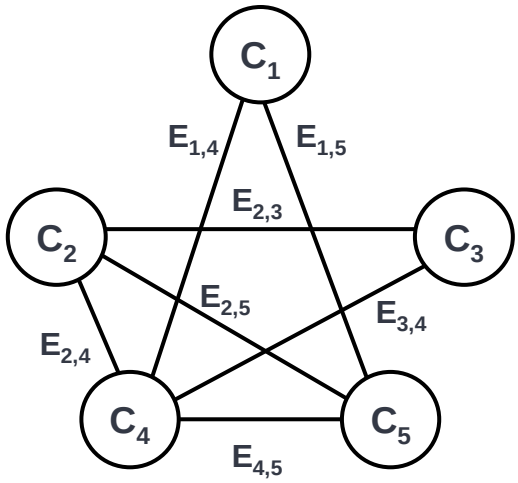

Figure 2.2: Abstract view of the backbone network of a Grid system in Fig. 1.2 (comprising of master cluster nodes alone).

2.2 Scheduling across Cluster Systems

The backbone network, in the computational Grid system, comprising of the master nodes of the clusters, form an arbitrary topology/graph $G = \langle C, E \rangle$, where C denotes the number of master nodes interconnected via E communication links, as illustrated in the Fig. 2.2. The master nodes and the links are assumed to be heterogeneous, that is, their respective speeds may not be identical. Thus, the edges have weights corresponding to the speeds of the links. We assume a uni-port communication model and that all the master nodes in the system have front-ends. This means that each master node can compute and communicate with another master node (to which it is connected directly via a link), simultaneously. We consider the load to originate at any node in the network and obtain a load distribution that minimizes the total processing time of the load. Without loss of

generality, we shall assume that all nodes in the system are capable of processing the load, that is, the required application to process the load is assumed to be available at all the nodes.

Wong et al [57] have proposed scheduling strategies for multiple divisible loads on linear daisy chain networks. Jingxi et al [58] have studied adaptive load distribution strategies for divisible load scheduling on resource unaware multi-level tree networks. England et al [59] have proven that the optimal solution to single-installment based divisible load scheduling problem on a arbitrary graph indeed occurs on a spanning tree of the graph, a multi-level tree. Yao and Bharadwaj [30] have studied the problem of scheduling divisible loads on arbitrary graphs assuming uni-port communication model. Their parallel distribution approach comprises of two stages. They first identify a minimum spanning tree (MST) for the network, and then dispatch the load on the MST. They propose two strategies namely, the resource-aware optimal load distribution (RAOLD) and RAOLD with optimal sequencing (RAOLD-OS). RAOLD uses the rule A in the literature [16] to obtain a reduced optimal tree, while RAOLD-OS uses the optimal sequencing [16] to determine the distribution sequence. Both algorithms guarantee optimal load distribution with RAOLD-OS always providing the minimum processing time. Further, Byrnes et al [60] have proved that finding the optimal spanning tree (the spanning tree that generates minimum total processing time) on the arbitrary network is NP-hard.

Therefore, one immediate question to address is which distribution strategy or

spanning tree(s) deliver efficient solutions for scheduling across cluster systems in a Grid environment. However, in the literature, there is no systematic comparative study of the performance of different spanning tree construction strategies for divisible load scheduling in a Grid environment. In Chapter 3, we present the distribution strategies for distributing divisible load across clusters in a CGS and in Chapter 6, we propose Resource Aware Sequential and Parallel Load Distribution (RASLD and RAPLD) strategies and compare their performance as well as those of spanning tree construction strategies.

Chapter 3

Load Distribution Strategies

The system model for a computational Grid system (CGS) was presented in the last chapter. In this chapter, we shall describe the load distribution strategies used in our algorithms for scheduling within as well as across clusters in a CGS. Communication delay is assumed to be negligible within clusters and the distribution strategy for such a system is described in Section 3.1 and the communication delay is considered for scheduling across clusters and the corresponding distribution strategies are detailed in Section 3.2.

In the DLT literature [16], in order to derive an optimal solution it was mentioned that it is necessary and sufficient that all the sinks that participate in the computation must stop at the same time instant; otherwise, load could be redistributed to improve the processing time. We use this optimality principle in the design of all our load distribution strategies for CGS.

3.1 Systems with no Communication Delays

The system model for scheduling within clusters, assuming a multi-port communication model, was described in Chapter 2. The timing diagram for load distribution in such a system is shown in Fig. 3.1, where there are N source and M sink nodes. The timing diagram represents the communication and computation times of the sources and sinks within the system, with the x-axis representing the time. From the timing diagram, we see that,

$$\sum_{i=1}^{N} \alpha_{i,j} w_j T_{\text{cp}} = \sum_{i=1}^{N} \alpha_{i,j+1} w_{j+1} T_{\text{cp}}, \;\; j = 1, ..., M-1 \tag{3.1}$$

As our objective is to determine a unique solution for the optimal fractions $\alpha_{i,j}$, we impose the following condition in our strategy. Let

$$\alpha_{i,j} = \alpha_j L_i \;, \quad i = 1, ..., N \;, \quad j = 1, .., M \tag{3.2}$$

This condition essentially assumes that each sink requests a load fraction that is proportional to the size of the load at the source. Moreover, each sink requests the same load fraction (percentage of total load) from each source. Without this condition, it may be noted that the system of equations is under-constrained and additional constraints are needed to obtain a unique solution. With this condition

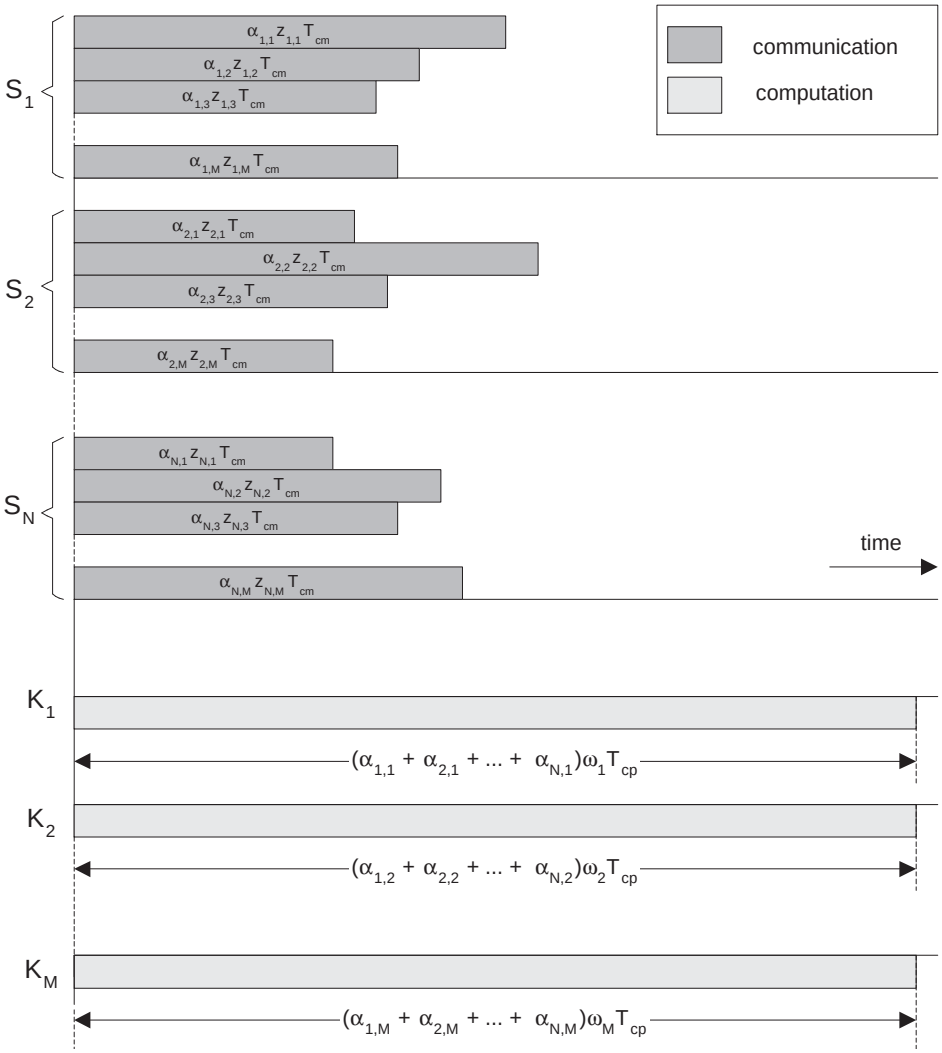

Figure 3.1: Timing diagram for the load distribution strategy with N sources and M sinks in an iteration within clusters.

(3.1) simplifies to

$$\sum_{i=1}^{N} \alpha_j L_i w_j = \sum_{i=1}^{N} \alpha_{j+1} L_i w_{j+1} \ , \quad j = 1, ..., M-1 \tag{3.3}$$

Using (3.3) together with the fact that $\sum_{i=1}^{M} \alpha_i = 1$, we have

$$\alpha_j = \frac{1}{w_j(\sum_{x=1}^{M} \frac{1}{w_x})} \ , \quad j = 1, ..., M \tag{3.4}$$

Hence, the fraction of load that K_j should request from S_i is derived as

$$\alpha_{i,j} = \frac{1}{w_j(\sum_{x=1}^{M} \frac{1}{w_x})} L_i \tag{3.5}$$

In real-life situations, there is always a limit to the amount of buffer space that a sink could render. Further, in a real-life environment, each node may be running multiple tasks such that it is required to share the available resources, hence there may be only a limited amount of buffer space that is allocated for processing particular loads at a given time. As a result, we are naturally confronted with the problem of scheduling divisible loads under buffer capacity constraints. Hence, if there is sufficient load in the system to completely consume a buffer at one of the sink nodes, the load fractions $\alpha_{i,j}$ that a sink K_j shall request from a source S_i has to be reduced by a factor Y, given by

$$Y = \min \left\{ \frac{B_j}{(\alpha_j L)} \right\} \tag{3.6}$$

Proposition 1: The factor Y defined in (3.6) ensures that at each iteration all the sinks that participate in processing the loads complete processing at the same time instant. □

Similar load distribution strategy is also used in [56] for off-line scheduling. But, in our strategy, we utilize the IBS algorithm in every iteration and attempt to fill up one or more sinks' buffer space. This load distribution strategy forms the basis of our schedulers for distributions within clusters. In every iteration we attempt to fill up one or more sinks' buffer space. If in an iteration, the remaining load is not enough to completely consume the buffer at a participating sink node, we use the distribution suggested by (3.5). In our strategies, when multiple sinks have identical buffer capacities, the buffer at the fastest sinks will be fully utilized.

3.2 Systems with Communication Delays

In this section, we propose load distribution strategies for scheduling across clusters assuming a uni-port communication model. The system model for such systems are detailed in Chapter 2. As with the distribution strategy for scheduling within clusters, here too all the processing nodes are assumed to start computing immediately upon receiving the load portions assigned to them. As mentioned earlier, England et al [59] have proven that the optimal solution to single-installment based divisible load scheduling problem on a arbitrary graph indeed occurs on a spanning tree of the graph, a multi-level tree. Hence, given an arbitrary graph, we shall first

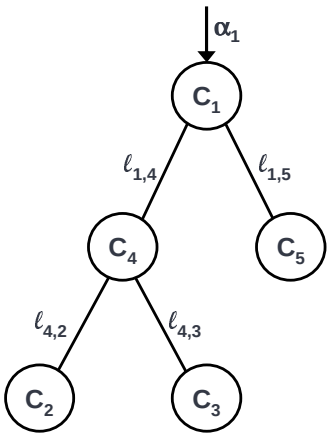

Figure 3.2: A spanning tree for the backbone network of a Grid system in Fig. 2.2 with a load (α_1) at the node C_1.

generate a spanning tree for it as detailed later in Chapter 6. Fig. 3.2 illustrates a spanning tree for the abstract Grid System in Fig. 2.2. Here, we assume that a spanning tree is generated for the given network, and propose two distribution strategies, namely sequential and parallel load distributions for distributing the load on that multi-level tree network. In the case of sequential distribution, the given spanning tree is reduced to a single-level tree by adding up the link delays from the root node to the processing nodes and in the case of parallel distribution it is achieved by recursively reducing the multi-level tree to a single-level tree.

The optimal sequencing theorem for a single-level tree network presented in [16] is applied at each single-level tree to determine the optimal sequence at that level. The optimal sequencing theorem states that

Theorem 1 (Optimal sequence): In a single-level tree network $\sum(x, i, m+1)$, in order to achieve minimum processing time, the sequence of load distribution by the parent node $C_{x,i}$ should follow the order in which the link speeds $(\frac{1}{z_{i,k}}, k = 1, 2, ...m)$

decrease.

This theorem provides a necessary and sufficient condition for a sequence of load distribution to be optimal. Thus, for an optimal solution, the load is distributed first through the fastest link, then through the next fastest link, and so on until the slowest link is assigned the last load fraction.

The sequential and parallel load distribution strategies are described in the following sections.

3.2.1 Sequential Distribution

In this section, we propose sequential load distribution strategy for distributing the divisible loads from the load originating node (root node) to other cluster coordinator (master) nodes in the system. In this strategy, the root node shall distribute the load to other master nodes in the system sequentially, after reducing the multi-level tree to a single-level tree systematically as follows. We shall consider all the nodes in a spanning tree network, compute the sum of link delays (communication delays) along the path from the root node to them, and derive a single-level tree with the computed sum as the link delay value for the link between the root node and that node, as shown in Fig. 3.3, arrange the nodes following the optimal sequence (*Theorem 1*) and determine the distribution. Then, when the root node distributes the load to other nodes, it shall sequentially distribute the load portions assigned to them also following the optimal sequence order, as illustrated in

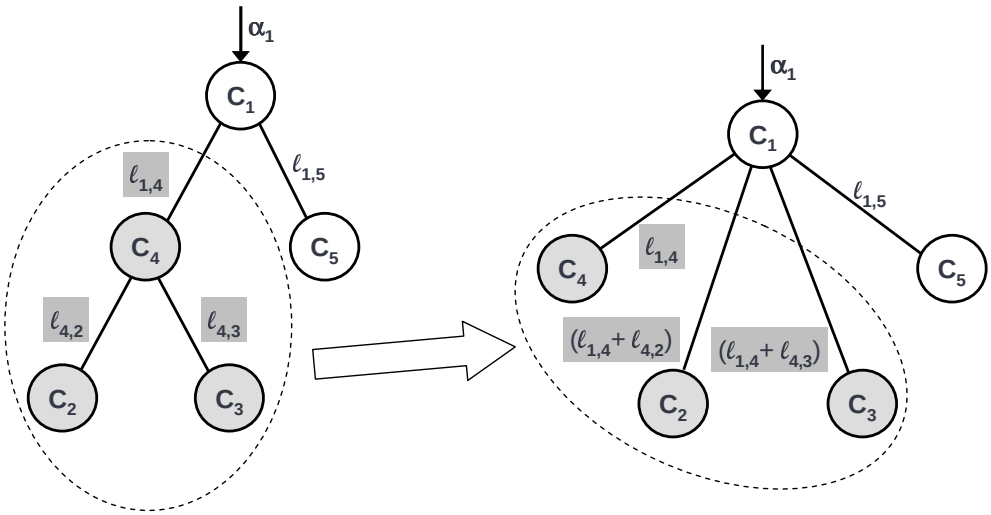

Figure 3.3: Reducing a multi-level tree to a single-level tree for sequential load distribution on the spanning tree in Fig. 3.2.

the timing diagram (Fig. 3.5). The timing diagram represents the communication and computation times of the root node and other cluster coordinator nodes in a CGS, with the x-axis representing the time.

In [16], the equivalent processor and link speeds for a single-level tree network in Fig. 3.4 is derived as

$$w_{0,(k+1,\ldots,k+r)} = \left(\frac{1}{1 + \Sigma_{u=k+2}^{k+r} \Pi_{v=u}^{k+r} f_v} \right) w_{0,(k+r)} \tag{3.7}$$

and

$$z_{0,(k+1,\ldots,k+r)} = \frac{\Sigma_{u=k+2}^{k+r} \{ (\Pi_{v=u}^{k+r}) z_{0,(u-1)} \} + z_{0,(k+r)}}{1 + \Sigma_{u=k+2}^{k+r} \Pi_{v=u}^{k+r} f_v} \tag{3.8}$$

where

$$f_v = \frac{w_{0,v} + z_{0,v}(T_{\mathrm{cm}}/T_{\mathrm{cp}})}{w_{0,(v-1)}} \tag{3.9}$$

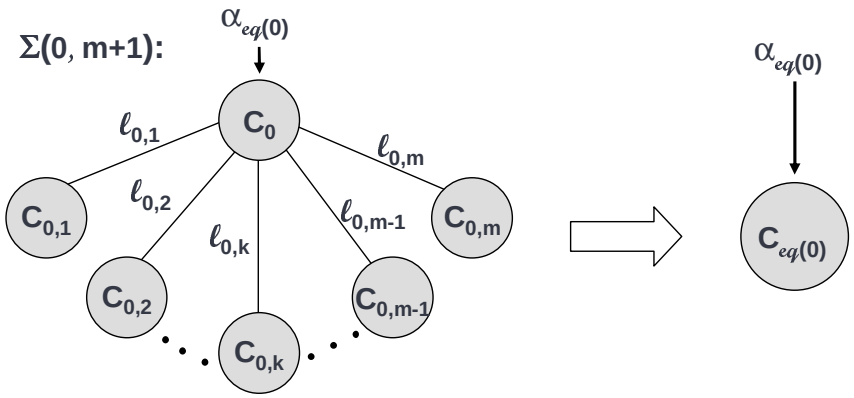

Figure 3.4: Processor equivalence for a single-level tree of the entire network.

From these, the processing time for the single-level tree network is derived as

$$\begin{aligned} T(\Sigma(0,m+1)) &= \alpha_{0,i} w_{0,i} T_{\text{cp}} \\ &= \left(\frac{\Pi_{v=1}^{m} f_v}{1 + \Sigma_{u=1}^{m} \Pi_{v=u}^{m} f_v} \right) w_{0,i} T_{\text{cp}} \\ &= w_{eq(0)} \cdot T_{\text{cp}} \end{aligned} \tag{3.10}$$

where

$$w_{eq(0)} = \left(\frac{\Pi_{v=1}^{m} f_v}{1 + \Sigma_{u=1}^{m} \Pi_{v=u}^{m} f_v} \right) w_{0,i} \tag{3.11}$$

Thus, as shown in Fig. 3.4, given a single-level tree network, the entire network could be replaced by a equivalent node $C_{eq(0)}$ whose processing speed is given by (3.11). The optimal load fractions that shall be distributed to the participating nodes is given by

$$\alpha_{0,k} = \alpha_{0,m} \, \Pi_{v=k+1}^{m} \, f_v \ , \quad k = 0, 1, ..., m-1 \tag{3.12}$$

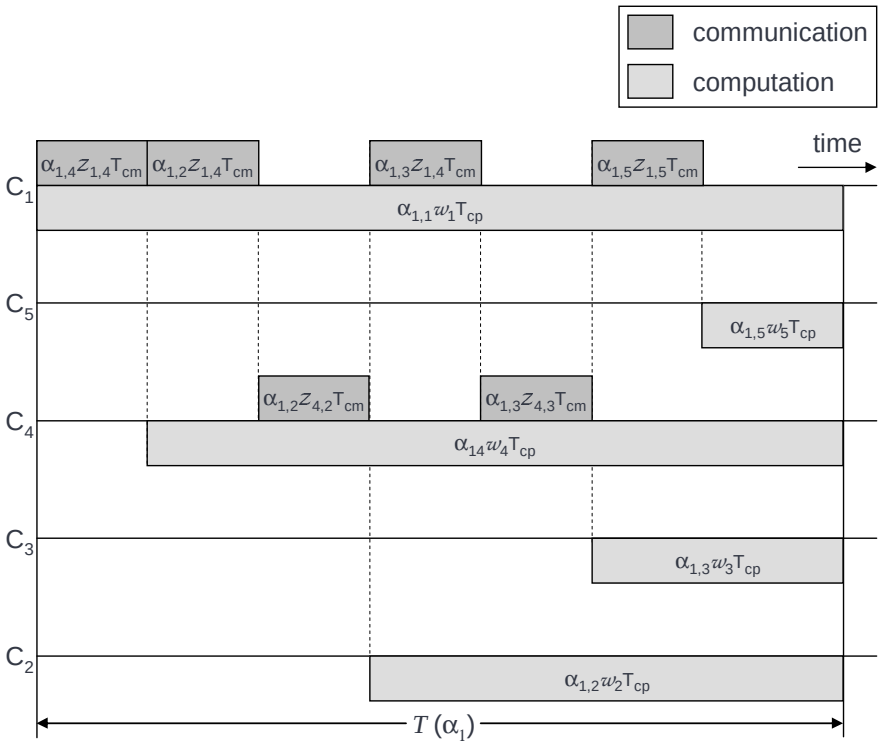

Figure 3.5: Timing diagram for the sequential load distribution strategy across cluster coordinator nodes.

where

$$\alpha_{0,m} = \frac{\alpha_{eq(0)}}{1 + \Sigma_{u=1}^{m} \Pi_{v=u}^{m} f_v} \tag{3.13}$$

and f_v is as defined in (3.9).

From the timing diagram (Fig. 3.5), we see that, at any given time, there is only one communication happening in the entire network. That is, even when there are load to be distributed and its front end is not utilized for any communication, the root node waits for the communications happening in the entire network to be completed. Hence, we call this strategy as *sequential distribution strategy.* Our parallel distribution strategy described in the next section, attempts to leverage on such idle periods to optimize the processing time.

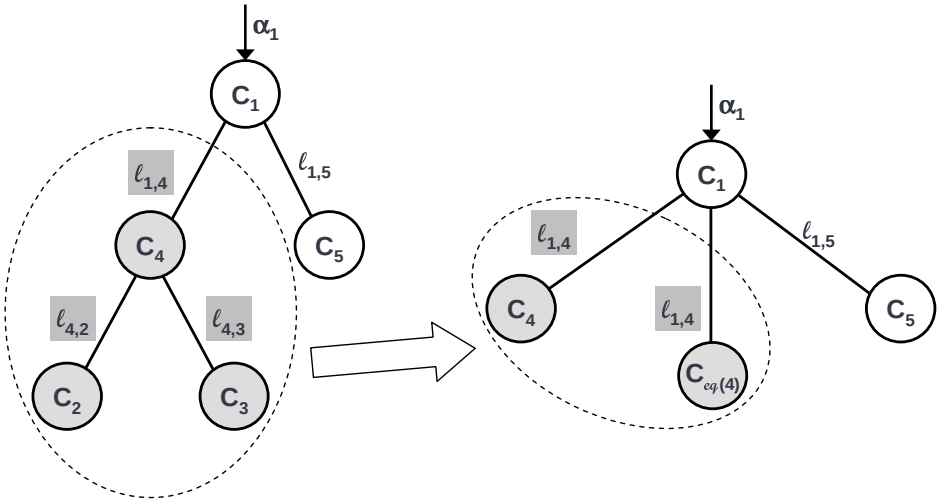

Figure 3.6: Reducing a multi-level tree to a single-level tree for parallel load distribution on the spanning tree in Fig. 3.2.

3.2.2 Parallel Distribution

In this section, we propose parallel load distribution strategy for distributing the divisible loads from the load originating node (root node) to other cluster coordinator nodes in the system. Here, we let the root and other parent nodes (the nodes that have child nodes) to distribute the loads to their children in a parallel manner as described below. We consider all the nodes in a spanning tree network; systematically reduce the given multi-level tree to a single-level tree by replacing the sub-trees with their equivalent node as in Fig. 3.6, arrange the nodes following the optimal sequence (*Theorem 1*) and determine the distribution. The parent node, while distributing the load to its children, shall distribute to its child node first and then to the sub-tree for which that child is a parent, and then proceed distributing to its next child and so on, also following the optimal sequence order as illustrated in the timing diagram (Fig. 3.8). The timing diagram represents

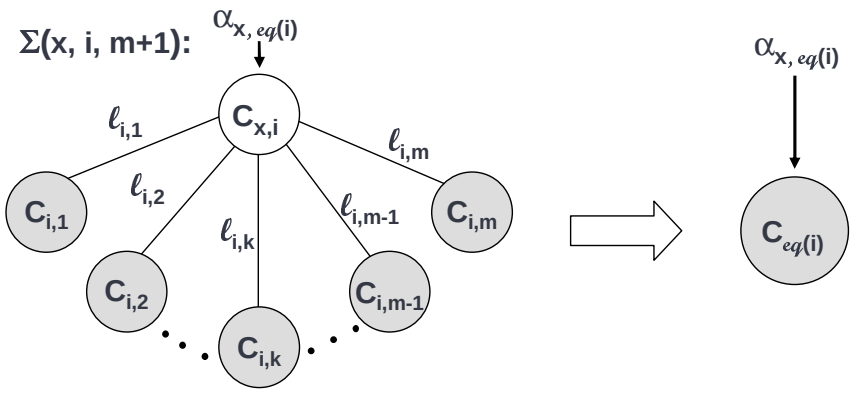

Figure 3.7: Processor equivalence for a single-level sub-tree.

the communication and computation times of the root node and other cluster coordinator nodes in a CGS, with the x-axis representing the time.

From Fig. 3.6, it shall be noted that C_4 acts as a control node for the communications between the root node C_1 and leaf nodes C_2 and C_3, relaying the loads assigned to them by the root node. Hence, this sub-tree is equivalent to a single-level tree network with a control node C_4.

In [16], the optimal processing time for a bus network with control processor (or control node) has been derived. A bus network is a special case of a single-level tree network where all the link delays ($z_{i,j}$) are identical. We generalize and extend the optimal processing time expression derived for the bus network to a single-level tree network in Fig. 3.7 as

$$T(\alpha_{x,eq(i)}) = (z_{i,1}\ T_{\mathrm{cm}}\ +\ w_{i,1}\ T_{\mathrm{cp}})\ \alpha_{i,1} \tag{3.14}$$

where

$$\begin{aligned} \alpha_{i,1} &= 1\,, \quad m = 1 \\ &= \frac{1}{1 + \Sigma_{i=1}^{m-1} \Pi_{j=1}^{i}(k_j)}\,, \quad m \geq 2 \end{aligned} \tag{3.15}$$

and

$$k_j = \frac{w_{i,j}\ T_{\text{cp}}}{z_{i,(j+1)}\ T_{\text{cm}}\ +\ w_{i,(j+1)}\ T_{\text{cp}}}\,, \quad 1 \leq j \leq m-1 \tag{3.16}$$

The equivalent processing capability value for the sub-tree could be computed using (3.14) as

$$w_{x,eq(i)} = \frac{T(\alpha_{x,eq(i)})}{T_{\text{cp}}} \tag{3.17}$$

Thus, nodes in a sub-tree whose parent is $C_{x,i}$ could be replaced with an equivalent node $C_{x,eq(i)}$ with a link to the parent node C_x with a delay value $z_{x,i}$ as shown in Fig. 3.7.

Given a multi-level tree we shall begin at the lowest level, arrange the nodes in an optimal sequence and recursively replace sub-trees with their equivalent nodes as computed in (3.17) till we reach the single-level tree with root node as the parent node, upon which we shall use (3.11), and determine the load distribution. While distributing the load to the children, the parent node shall follow the optimal sequencing order, inflate the equivalent nodes (if any) and optimally distribute the load assigned among the nodes that formed that equivalent node, as given by the

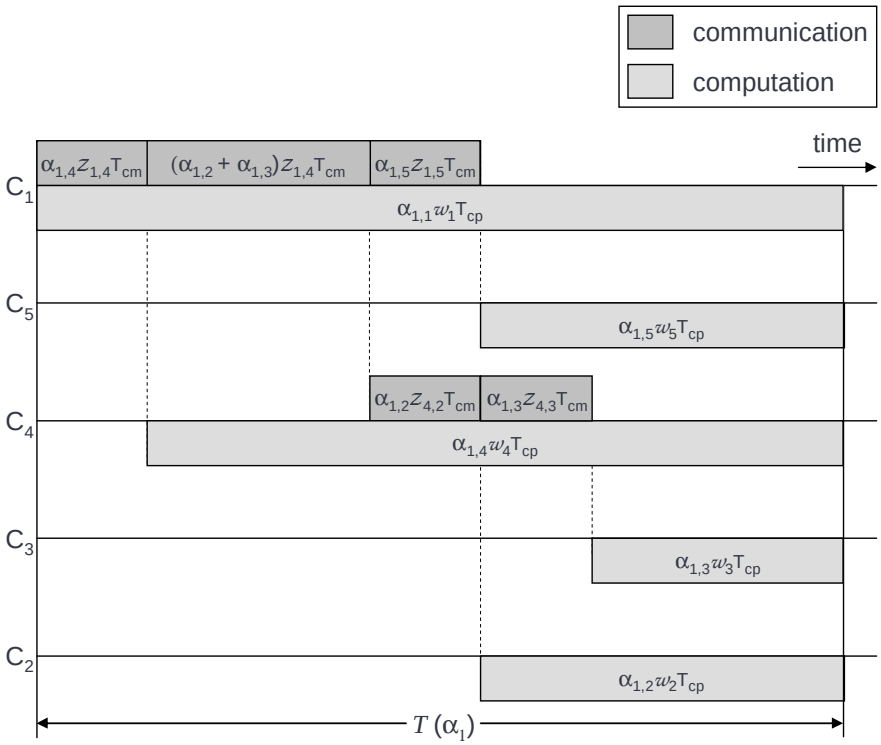

Figure 3.8: Timing diagram for the parallel load distribution strategy across cluster coordinator nodes.

following equations:

$$\alpha_{i,k} = \alpha_{i,m} \ \Pi_{v=k+1}^{m} \ f_v \ , \quad k = 0, 1, ..., m-1 \tag{3.18}$$

where

$$\alpha_{i,m} = \frac{\alpha_{x,eq(i)}}{1 + \Sigma_{u=1}^{m} \Pi_{v=u}^{m} f_v} \tag{3.19}$$

and f_v is as defined in (3.9) and $\alpha_{x,eq(i)}$ is the load assigned to $C_{x,eq(i)}$ by its parent node.

From the timing diagram (Fig. 3.8), we see that, in this distribution strategy, the root node C_1 first distributes the load assigned to C_4 as well as its sub-tree

(nodes C_2 and C_3) before proceeding to distribute to its other child C_5. Then, while the node C_4 is distributing the load to its children C_2 and C_3, the root node C_1 distributes the load to its other child C_5. Thus, in this strategy the parent nodes in the system distribute the load to their children concurrently after receiving the load portions from their parent nodes. Hence, we call this strategy as *parallel distribution strategy.* Also, C_4 starts computing as soon as it received its load portion from C_1, while continuing to receive the load portions assigned to its children from its parent node.

Chapter 4

Scheduling Strategies for Non-time Critical Loads

Strategies for distributing the loads both within and across cluster nodes are detailed in the last chapter. In this chapter, we shall propose dynamic and adaptive scheduling strategies for systems having non-time critical loads and finite buffer capacity constrained sink nodes within clusters. We consider two environments, where the buffer availability at sink nodes

- remain constant over time
- vary over time

and propose suitable strategies for them following the distribution strategies presented in Chapter 3. However, under deadline driven processing requirements, the

number of loads that can be admitted by the system needs to be restricted and it is discussed in Chapter 5.

Here, we consider scheduling when the loads arrive at arbitrary times to the cluster system for processing as well as when the total amount of loads to be processed exceeds the currently available buffer capacities. In a real-life system, the number of loads to be processed may vary over time and also demand for processing may arise at any time. Thus, it will be difficult to estimate a priori the maximum amount of load that may be in the system at any time. Under such conditions, a feasible schedule may not exist unless the sink nodes allow their buffers to be reclaimed after a given load is processed. This means that, after processing a given load, the sinks shall make their buffer available for subsequent processing. Thus in order to handle the situation wherein sources demand processing at various time instants, dynamic scheduling strategies needs to be designed in such a way that sinks continue to render their available buffers to the sources.

4.1 Dynamic IBS Algorithms

In the DLT literature [25], it was mentioned that for an optimal scheduling solution, it is necessary and sufficient if all the sinks that participate in the computation stops at the same time instant, else the loads could be redistributed to improve the processing time. The optimality principle stated in the DLT literature was used and load fractions that a sink K_j shall receive from the source S_i was derived in

the modified IBS algorithm [56] for systems with pre-specified buffer constraints.

The modified IBS algorithm recursively invokes IBS algorithm [34] and employs a "push-based" strategy. In this scheme, a source node identifies potential sinks (with knowledge about the available resources at the sinks), computes the schedule and communicates it to other source nodes. Upon receiving this schedule information, all the source nodes send their load portions to the respective sink nodes. Although this algorithm recursively attempts to fill up one or more sinks' buffer space at every iteration, it is basically an offline algorithm. In this scheme, when a sink's buffer is completely filled up, that sink is not considered for scheduling in the subsequent iterations.

The modified IBS algorithm exhibits finite convergence. But, it does not consider real life situations, where the buffer capacities at sink nodes vary over time and the loads to be processed may arrive at arbitrary times to the system.

For scheduling in dynamic environments, optimal load fractions must be recomputed at the completion of every iteration based on the total load in the system. This process shall continue until all of the loads are processed. Thus, the load requesting by the sinks and processing are on-line in the sense that the IBS algorithm is invoked to recompute the load distribution depending on the number of sources and their respective load sizes, after the sinks complete processing the loads requested by them earlier. Further, it shall be noted that the buffer space availability (depending on workload characteristics) in sinks does not have an affinity towards any source. Thus, if no other sources demand processing, then the entire buffer is

allocated to the demanding source. Since the IBS strategy is invoked for recomputing the loads at the end of every iteration and dynamic arrival of loads are also considered, we refer to this algorithm as the Dynamic IBS algorithm hereafter.

The Dynamic IBS algorithm ensures that at any iteration all the sinks stop computing at the same time instant. Hence, the optimal processing time for the q^{th} iteration is given by,

$$T^{(q)} = \sum_{i=1}^{N} \alpha_{i,j} w_j T_{\text{cp}} \tag{4.1}$$

where the values for $\alpha_{i,j}$ are the values for that iteration. The total load processed in an iteration is given by

$$\sum_{i=1}^{N} \sum_{j=1}^{M} \alpha_{i,j} \tag{4.2}$$

Hence, the time taken to process a unit load is given by

$$T_{\text{ul}} = \frac{\sum_{i=1}^{N} \alpha_{i,j} w_j T_{\text{cp}}}{\sum_{i=1}^{N} \sum_{j=1}^{M} \alpha_{i,j}} \tag{4.3}$$

The optimal processing time for the existing load in the system is given by

$$T_{\text{opt}} = T_{\text{ul}} \cdot \sum_{i=1}^{N} L_i \ , \ \forall S_i \in X_{\text{now}} \tag{4.4}$$

and the optimal processing time for the total load in the system including the newly arrived sources that are being considered is given by (4.4) with $S_i \in \{X_{\text{now}} \cup X_{\text{new}}\}$. It may be noted that the optimal processing time given by (4.4) is governed by the product of the total load in the system and the optimal time taken to process

a unit load. From (4.4), it is seen that as long as there are no new sinks and all sinks allow their declared buffer sizes to be reclaimed, the total processing time is directly proportional to the total load. So, if the total load increases the processing time also increases proportionately.

The new set of loads and the unprocessed loads from the existing sources are considered together for scheduling at the end of every iteration, that is, after the current processing is completed. In the absence of any new sources, the optimal time for processing the existing sources in the system approaches the distribution derived in the Modified IBS algorithm described in [56].

4.1.1 Time-invariant Buffer Environments

Here, we assume that neither the buffer sizes declared by the sinks vary over time nor any new sinks are added to the system. Dynamic IBS algorithm for time-invariant buffer environments at the coordinator and the sink nodes are presented in Fig. 4.1 and 4.2 respectively. Since, there are M sinks in the system, the complexity of this algorithm is $O(M)$.

Example 4.1 clarifies the working principle of the Dynamic IBS algorithm for time-invariant buffer environment. The sink speed ($\frac{1}{w_j}$) parameters for this example are derived from the STAR experiments conducted at BNL [51].

Example 4.1**:**

Let us consider a system with four sources and four sinks, with parameters $w_1 =$

Initial state:

$I = \{1, 2, ...N\}$, $J = \{1, 2, ...M\}$, $q = 0$, $T^{(0)} = 0$

$\{\ B_j^{(0)} = B_j\ ;\quad \alpha_j = 1/(w_j \sum_{x=1}^{M} \frac{1}{w_x})\ \}$, $\forall K_j$, $j \in J$

Broadcast (α_j) values to all the Sink nodes.

Step 1: Determine $T^{(q+1)}$:

If $(X_{\text{new}} \neq \emptyset)$ $\{\ X_{\text{now}} = X_{\text{now}} \cup X_{\text{new}}\ ;\quad X_{\text{new}} = \emptyset\ \}$

If $(X_{\text{now}} \neq \emptyset)$ $\{$

$L = \sum_{i=1}^{N} L_i$, $\forall S_i \in X_{\text{now}}$, $i \in I$

$Y = \min\{B_j/(\alpha_j L)$, $\forall K_j$, $j \in J\}$

If $(Y > 1)$ $\{Y = 1\}$

$T^{(q+1)} = Y \alpha_j L\ w_j\ T_{\text{cp}}$, *for any* K_j , $j \in J$

Broadcast the schedule information $(Y, (T + T^{(q+1)}), (L_i, \forall S_i \in X_{\text{now}}, i \in I))$ to all the Sink nodes.

Step 2: Update the amount of load remaining to be processed:

Wait till $(T + T^{(q)})$.

$q = q + 1$

$L_i = L_i - Y \alpha_j^{(q)} L_i, \forall S_i \in X_{\text{now}}, i \in I$

$\{$ If $(L_i = 0)$ $\{X_{\text{now}} = X_{\text{now}} - \{S_i\}\}$ $\}$, $\forall S_i \in X_{\text{now}}$, $i \in I$ $\}$

Go to **Step 1**.

Figure 4.1: Pseudo code describing the workings of the Dynamic IBS algorithm for time-invariant buffer environment at the coordinator node C_s.

Initial state:

$q = 0$, $T^{(0)} = 0$

Receive (α_j) value from the Coordinator node.

Step 1: Wait till the previous iteration is completed:

Wait till $(T + T^{(q)})$.

Step 2: Compute Load amounts to be processed:

Receive the schedule information $(Y, (T + T^{(q+1)}), (L_i, \forall S_i \in X_{\text{now}}, i \in I))$ from the Coordinator node.

$q = q + 1$

$\alpha_{i,j}^{(q)} = Y \alpha_j L_i$, $\forall S_i \in X_{\text{now}}$, $i \in I$

Step 3: Schedule the loads from Source Nodes:

Request, receive and process the load fractions $(\alpha_{i,j}^{(q)})$ from the Source Nodes $S_i \in X_{\text{now}}$.

Go to **Step 1**.

Figure 4.2: Pseudo code describing the workings of the Dynamic IBS algorithm for time-invariant buffer environment at the sink nodes.

Table 4.1: Sink and Source node parameters for Example 4.1.

Sink nodes	**Parameter**	
	Inverse of computing speed (w_j)	**Buffer capacity (B_j)**
Sink node 1 (K_1)	1.11×10^{-9}	6
Sink node 2 (K_2)	6.25×10^{-10}	5
Sink node 3 (K_3)	5.00×10^{-10}	2
Sink node 4 (K_4)	3.57×10^{-10}	3
Source nodes	**Parameter**	
	Load Size (L_i)	**Load arrival time**
Source node 1 (S_1)	5	0 sec
Source node 2 (S_2)	2	0 sec
Source node 3 (S_3)	3	0 sec
Source node 4 (S_4)	9	4×10^3 sec

1.11×10^{-9}, $w_2 = 6.25 \times 10^{-10}$, $w_3 = 5.00 \times 10^{-10}$, $w_4 = 3.57 \times 10^{-10}$, and $T_{\text{cp}} = 6.52 \times 10^{12}$sec/load. We let the sources have loads $L_1 = 5$, $L_2 = 2$, $L_3 = 3$ and $L_4 = 9$ units, respectively. We let the sinks having buffer capacities $B_1 = 6$, $B_2 = 5$, $B_3 = 2$, and $B_4 = 3$, respectively. We assume that the loads L_1, L_2, and L_3 arrives at $t = 0$ seconds and L_4 arrives at $t = 4 \times 10^3$ seconds. The sink and source node parameters are summarized in the Table 4.1.

Using the algorithm presented in Fig. 4.1 and 4.2, we have the values for $\alpha_{i,j}^{(q)}$ as shown in Table 4.2. The unutilized buffer space in all the iterations are shown in the last column of Table 4.2. From, these results, we observe that the buffer of K_3 is fully consumed at the first and second iterations. At the final iteration, the remaining load is insufficient to completely fill up the buffer of any of the

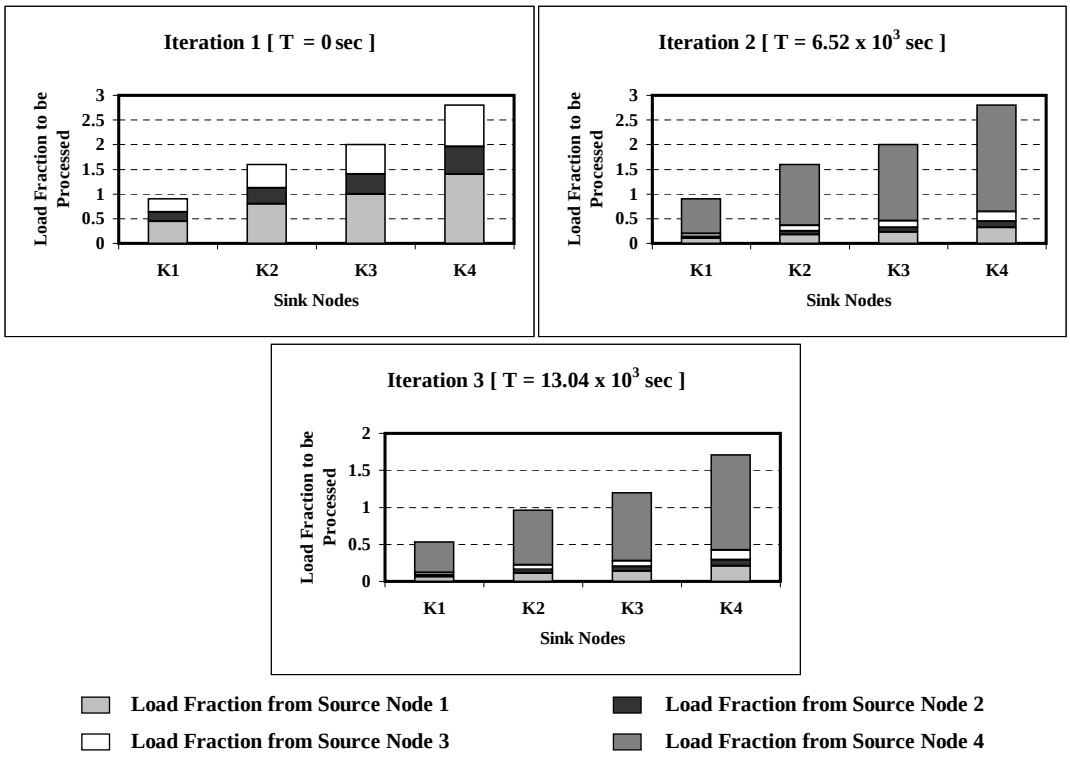

Figure 4.3: Performance of Dynamic IBS algorithm in time-invariant buffer environment.

Table 4.2: Load fraction and buffer utilization values for Example 4.1.

$\mathbf{q = 1}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$		$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{1})}$	$\mathbf{B_j^{(1)}}$
K_1	0.45	0.18	0.27		0.90	5.10
K_2	0.80	0.32	0.48		1.60	3.40
K_3	1.00	0.40	0.60		2.00	0.00
K_4	1.40	0.56	0.84		2.80	0.20
$\mathbf{q = 2}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$	$\mathbf{S_4}$	$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{2})}$	$\mathbf{B_j^{(2)}}$
K_1	0.10	0.04	0.06	0.70	0.90	5.10
K_2	0.18	0.07	0.11	1.24	1.60	3.40
K_3	0.23	0.09	0.14	1.54	2.00	0.00
K_4	0.32	0.13	0.19	2.16	2.80	0.20
$\mathbf{q = 3}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$	$\mathbf{S_4}$	$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{3})}$	$\mathbf{B_j^{(3)}}$
K_1	0.06	0.02	0.04	0.41	0.53	5.47
K_2	0.11	0.05	0.06	0.74	0.96	4.04
K_3	0.14	0.06	0.08	0.92	1.20	0.80
K_4	0.21	0.08	0.13	1.29	1.71	1.29

sinks. The distribution suggested by the values $\alpha_{i,j}$ in the Table 4.2 are used by the sinks. The values of $\alpha_{i,j}$ for the three iterations are computed at $t = 0,\ 6.52 \times 10^3,$ and 13.04×10^3 seconds, respectively. The total processing time for processing all the four loads is $t = 17.02 \times 10^3$ seconds. The load fractions requested (from the source nodes) and processed by the sink nodes at various iterations are shown in Fig. 4.3. It is seen that, because of the new source S_4, the processing time for the other sources in the system is stretched from $t = 8.93 \times 10^3$ seconds to $t = 17.02 \times 10^3$ seconds. The above raise in the overall processing time is acceptable if the submitted loads are not driven by any deadline requirements.

4.1.2 Predictable Time-varying Buffer Environments

The algorithm presented in the Section 4.1.1 is suitable for systems wherein there are dynamic load arrivals and the buffer capacities available at sink nodes are time-invariant. But, in certain real-life situations, the buffer space availability at sink nodes may also vary over time and these variations may be known a priori. We consider such systems in this section and propose a dynamic scheduler for them. The scheduling strategy is such that the coordinating node shall first obtain the information about the available buffer capacities at other sinks at various time instants, their computing speeds, and the size of the loads from the sources. The coordinating node shall then compute and notify each sink on the optimum load fractions that are to be requested from each source. These information can be easily communicated using any of the standard or customized communication protocols

without incurring any significant communication overhead. The sources, upon knowing the amount of loads that they should give to each sink, shall send their loads to all sinks simultaneously and the sinks shall start processing as they receive the loads from the sources.

In the algorithm proposed, in this section, for systems where the buffer spaces available at sinks vary over time, we assume that the buffer space variation over time at sinks is known a priori. If the time t at which the buffer sizes vary at any of the sink node is earlier than the iteration completion time ($T^{(q)}$), then the load fractions $\alpha_{i,j}$ that the sinks should request from a source S_i shall be computed as

$$\alpha_{i,j} = \alpha_{i,j} * \frac{t}{T^{(q)}} \tag{4.5}$$

to ensure that buffer sizes at sink nodes do not change during an iteration. If the buffer spaces do not vary at the end of an iteration, then they are allowed to be reclaimed fully after processing is completed, so as to enable scheduling more amount of loads. The optimal load fractions are recomputed at the completion of every iteration, based on the total load in the system. The algorithm for the coordinator and the sink nodes are presented in Fig. 4.4 and 4.5 respectively. Since, there are M sinks in the system, the complexity of this algorithm is $O(M)$.

Example 4.2 clarifies the working principle of the Dynamic IBS algorithm for predictable time-varying buffer environment. The sink speed ($\frac{1}{w_j}$) parameters for this example are derived from the STAR experiments conducted at BNL [51].

Initial state:

$I = \{1, 2, ...N\}$, $J = \{1, 2, ...M\}$, $q = 0$, $t = 0$, $T^{(0)} = 0$

Step 1: Determine $\alpha_j^{(q+1)}$ & $T^{(q+1)}$:

If ($X_{\text{new}} \neq \emptyset$) { $X_{\text{now}} = X_{\text{now}} \cup X_{\text{new}}$; $X_{\text{new}} = \emptyset$ }

If ($X_{\text{now}} \neq \emptyset$) {

If ($T = t$) { $P_{\text{now}} = P_{\text{all}}$

$B_j = B_{j,t}$, $\forall K_j$, $j \in J$

{ If ($B_j = 0$) $P_{\text{now}} = P_{\text{now}} - K_j$ } , $\forall K_j$, $j \in J$

$\alpha_j = 1/(w_j \sum_{x=1}^{M} \frac{1}{w_x})$, $\forall K_j \in P_{\text{now}}$, $j \in J$

$t = t_{\text{next}}$ }

$L = \sum_{i=1}^{N} L_i$, $\forall S_i \in X_{\text{now}}$, $i \in I$

$Y = \min\{B_j/(\alpha_j L)$, $\forall K_j \in P_{\text{now}}$, $j \in J\}$

If ($Y > 1$) $\{Y = 1\}$

$T^{(q+1)} = Y \alpha_j L \, w_j \, T_{\text{cp}}$, *for any* K_j , $j \in J$

Broadcast the schedule information (Y, α_j, $(T + T^{(q+1)})$, $(L_i, \forall S_i \in X_{\text{now}}, i \in I)$) to all the Sink nodes.

Step 2: Update the amount of load remaining to be processed:

Wait till $(T + T^{(q)})$.

$q = q + 1$

$L_i = L_i - Y \alpha_j L_i$, $\forall S_i \in X_{\text{now}}$, $i \in I$

{ If ($L_i = 0$) $\{X_{\text{now}} = X_{\text{now}} - \{S_i\}\}$ }, $\forall S_i \in X_{\text{now}}$, $i \in I$ }

Go to **Step 1**.

Figure 4.4: Pseudo code describing the workings of the Dynamic IBS algorithm for predictable time-varying buffer environment at the coordinator node C_s.

Initial state:
$q = 0$, $T^{(0)} = 0$

Step 1: Wait till the previous iteration is completed:
Wait till $(T + T^{(q)})$.

Step 2: Compute Load amounts to be processed:
Receive the schedule information (Y, α_j, $(T + T^{(q+1)})$, $(L_i, \forall S_i \in X_{\text{now}}, i \in I)$) from the Coordinator node.
$q = q + 1$
$\alpha_{i,j}^{(q)} = Y\alpha_j L_i$, $\forall S_i \in X_{\text{now}}$, $i \in I$

Step 3: Schedule the loads from Source Nodes:
Request, receive and process the load fractions $(\alpha_{i,j}^{(q)})$ from the Source Nodes $S_i \in X_{\text{now}}$.
Go to **Step 1**.

Figure 4.5: Pseudo code describing the workings of the Dynamic IBS algorithm for predictable time-varying buffer environment at the sink nodes.

Example 4.2**:**

Let us suppose that there are four sources with loads to be processed and there are four sinks that can process these loads. Let the speed parameters be $w_1 = 1.11\times10^{-9}$, $w_2 = 6.25\times10^{-10}$, $w_3 = 5.00\times10^{-10}$ and $w_4 = 3.57\times10^{-10}$, respectively. Let $T_{\text{cp}} = 6.52 \times 10^{12}$sec/load. Let the buffer capacities at sinks at time $t = 0$ seconds be $B_1 = 6$, $B_2 = 5$, $B_3 = 0$, and $B_4 = 3$; at time $t = 5 \times 10^3$ seconds be $B_1 = 2$, $B_2 = 3$, $B_3 = 2$, and $B_4 = 3$; and at time $t = 1 \times 10^4$ seconds be $B_1 = 0$, $B_2 = 1$, $B_3 = 1$, and $B_4 = 1$ respectively. We let the four sources to have loads $L_1 = 5$, $L_2 = 2$, $L_3 = 3$ and $L_4 = 4$ units, respectively. Let loads L_1 to L_3 arrive at $t = 0$ seconds, and load L_4 arrive at $t = 8 \times 10^3$ seconds. The sink and source node parameters are summarized in the Table 4.3.

Table 4.3: Sink and Source node parameters for Example 4.2.

Sink nodes	**Parameter**	
	Inverse of computing speed (w_j)	**Buffer capacity (B_j)**
Sink node 1 (K_1)	1.11×10^{-9}	6 [at 0 sec] 2 [at 5×10^3 sec] 0 [at 10×10^3 sec]
Sink node 2 (K_2)	6.25×10^{-10}	5 [at 0 sec] 3 [at 5×10^3 sec] 1 [at 10×10^3 sec]
Sink node 3 (K_3)	5.00×10^{-10}	0 [at 0 sec] 2 [at 5×10^3 sec] 1 [at 10×10^3 sec]
Sink node 4 (K_4)	3.57×10^{-10}	3 [at 0 sec] 3 [at 5×10^3 sec] 1 [at 10×10^3 sec]
Source nodes	**Parameter**	
	Load Size (L_i)	**Load arrival time**
Source node 1 (S_1)	5	0 sec
Source node 2 (S_2)	2	0 sec
Source node 3 (S_3)	3	0 sec
Source node 4 (S_4)	4	8×10^3 sec

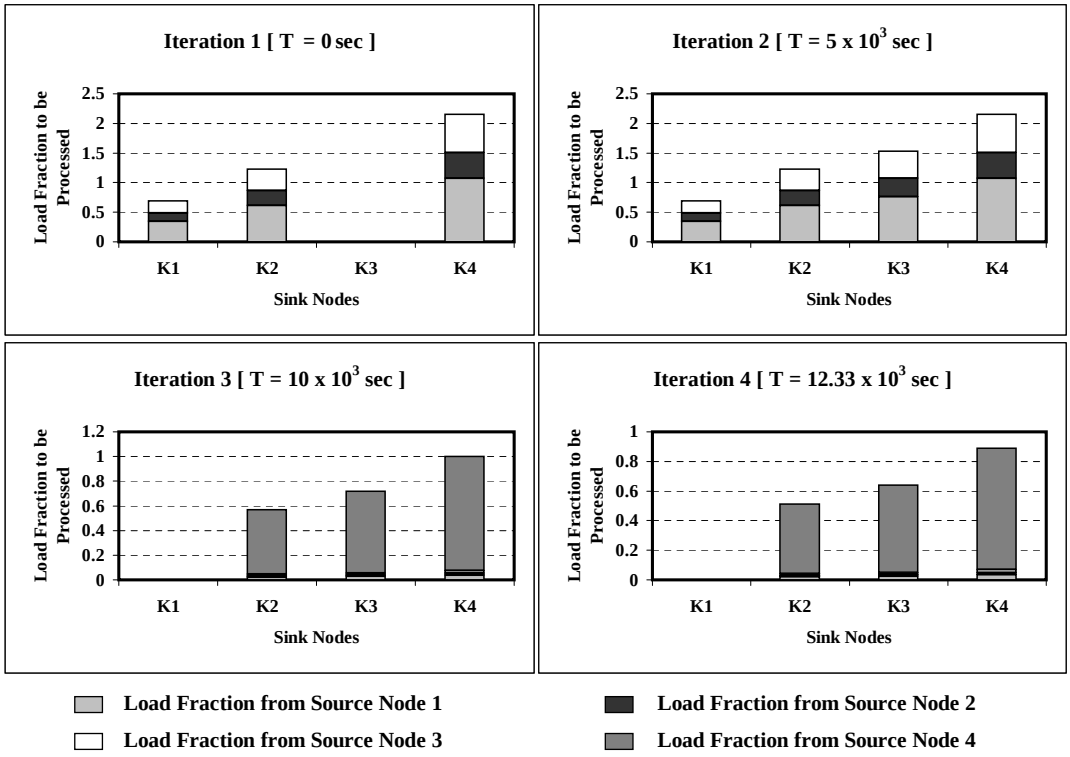

Figure 4.6: Performance of Dynamic IBS algorithm in predictable time-varying buffer environment.

Table 4.4: Load fraction and buffer utilization values for Example 4.2.

$\mathbf{q = 1}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$		$\sum \alpha_{\mathbf{i,j}}^{\mathbf{(1)}}$	$\mathbf{B_j^{(1)}}$
K_1	0.345	0.138	0.207		0.69	5.31
K_2	0.615	0.246	0.369		1.23	3.77
K_4	1.075	0.430	0.645		2.15	0.85
$\mathbf{q = 2}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$		$\sum \alpha_{\mathbf{i,j}}^{\mathbf{(2)}}$	$\mathbf{B_j^{(2)}}$
K_1	0.345	0.138	0.207		0.69	1.31
K_2	0.615	0.246	0.369		1.23	1.77
K_3	0.765	0.306	0.459		1.53	0.47
K_4	1.075	0.430	0.645		2.15	0.85
$\mathbf{q = 3}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$	$\mathbf{S_4}$	$\sum \alpha_{\mathbf{i,j}}^{\mathbf{(3)}}$	$\mathbf{B_j^{(3)}}$
K_2	0.022	0.009	0.013	0.526	0.57	0.43
K_3	0.027	0.011	0.016	0.666	0.72	0.28
K_4	0.038	0.015	0.023	0.924	1.00	0.00
$\mathbf{q = 4}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$	$\mathbf{S_4}$	$\sum \alpha_{\mathbf{i,j}}^{\mathbf{(4)}}$	$\mathbf{B_j^{(4)}}$
K_2	0.020	0.008	0.012	0.471	0.51	0.49
K_3	0.024	0.010	0.014	0.591	0.64	0.36
K_4	0.034	0.013	0.021	0.822	0.89	0.11

Using the algorithm in Fig. 4.4 and 4.5, we have the values for $\alpha_{i,j}^{(q)}$ as shown in Table 4.4. The unutilized buffer space in all the iterations are shown in the last column of Table 4.4. From these results, we observe that none of the buffers are fully utilized in iterations 1 and 2, because of buffer space variations at sinks over time. At iteration 3, a new source S_4 is accepted for processing. Note that this source is considered at $t = 1 \times 10^4$ seconds, although it arrived at $t = 8 \times 10^3$ seconds. Also, in this iteration, buffer of K_4 is fully utilized and at the final iteration, the remaining load is insufficient to completely fill up the buffer at any of the sinks. The distribution suggested by the values $\alpha_{i,j}$ in the Table 4.4 shall be used by the sinks. The values of $\alpha_{i,j}$ for iteration 1 to 4 are computed at $t = 0,\ 5\times10^3,\ 10\times10^3,$ and 12.33×10^3 seconds, respectively. The total processing time for processing all the four loads is $t = 14.4 \times 10^3$ seconds. From this example, it is seen that, because of the new source S_4 and the buffer space variations at the sinks, the processing time for the other sources in the system is stretched to $t = 14.4 \times 10^3$ seconds. The load fractions requested (from the source nodes) and processed by the sink nodes at various iterations are shown in Fig. 4.6. The above increase in overall processing time is acceptable if the submitted loads are not driven by any time critical requirements.

4.2 Adaptive IBS Algorithm

Dynamic IBS algorithms that consider dynamic load arrivals in time-invariant and predictable time-varying buffer environments were presented in the previous section. In this section, we propose an Adaptive IBS algorithm for real-life scenarios, wherein the actual buffer variations at sink nodes are not known a priori. Under such conditions, we propose that the sinks estimate the amount of buffer space that it could offer for scheduling in the next iteration and communicate it to the coordinator node and that during an iteration (while processing the received loads) the buffer spaces available at the sink nodes do not vary. A buffer estimation strategy is described in the Section 4.2.1. With this information, the coordinator node shall generate an initial schedule satisfying the resource constraints.

In the proposed algorithm, the load fractions are calculated based on the estimated buffer availabilities at the sinks. But, at the start of the next iteration, the actual buffer availabilities at the sinks may be different from the estimated values. As long as the load fractions assigned to each sink node by the coordinator node C_s is less than or equal to the actual buffer availabilities at those sink nodes, the sink nodes can request for the load fractions assigned to them from the sources. But, if the buffer available at a sink is less than the load fraction assigned to it, then it could not process the excess load that has been assigned to it. Hence, those sinks

shall recompute the load fraction to be received from the sources as

$$\alpha_{i,j} \;=\; \hat{\alpha}_{i,j} \;*\; \frac{B_j}{\sum_{i=1}^{N} \hat{\alpha}_{i,j}} \tag{4.6}$$

In addition to requesting these load fractions from the sources, the sink node also has to communicate the actual amount of load that it has received from the sources to the coordinator node. This ensures that the coordinator node will take into consideration the actual amount of loads that remain at the sources for processing, while computing the load fractions for the next iteration. This information can be piggy backed along with the estimated buffer availability at the sink nodes that all sink nodes communicate to the coordinator node. In the proposed strategy, all the sink nodes in the system (irrespective of whether it completes processing earlier or does not participate in that iteration) waits for all the sink nodes to complete their processing in an iteration (that is, for the time $T^{(q)}$) before requesting the loads from the sources again.

The optimal load fractions for the $(q+1)^{\text{th}}$ iteration shall be estimated by the coordinator node while the sinks process the load for the q^{th} iteration, based on the total amount of load that remains to be processed. This process shall continue until all of the loads are processed. Flowchart for the scheduler at the coordinator and the sink nodes are presented in Fig. 4.7 and 4.8 respectively. The pseudo code for the coordinator and the sink nodes are presented in Fig. 4.9 and 4.10 respectively. Since, there are M sinks in the system, the complexity of this algorithm is $O(M)$.

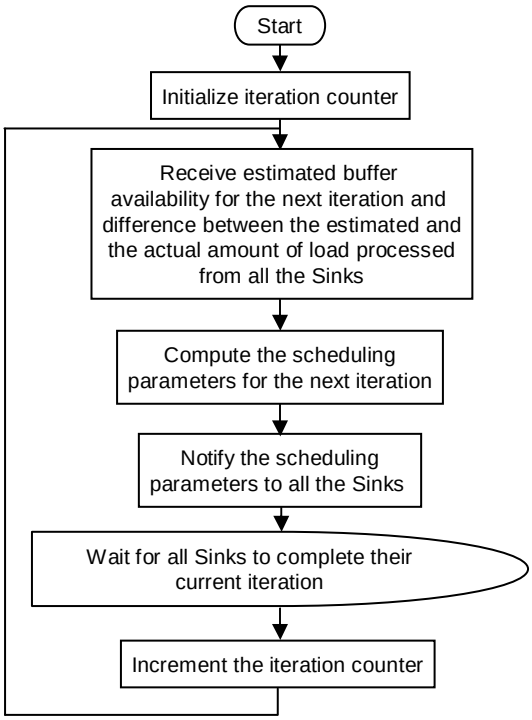

Figure 4.7: Flowchart for the workings of the Adaptive IBS algorithm at the coordinator node.

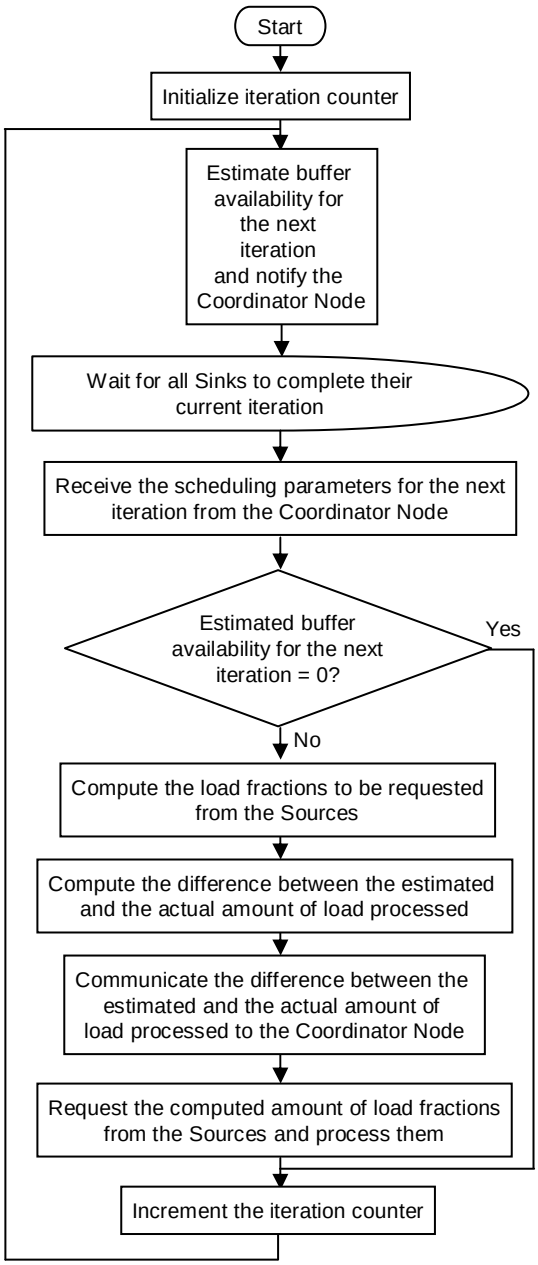

Figure 4.8: Flowchart for the workings of the Adaptive IBS algorithm at the sink nodes.

Initial state:

$I = \{1, 2, ...N\}$, $J = \{1, 2, ...M\}$, $q = 0$, $T^{(0)} = 0$

Step 1: Determine the buffer availability at Sink nodes:

If ($X_{\text{new}} \neq \emptyset$) { $X_{\text{now}} = X_{\text{now}} \cup X_{\text{new}}$; $X_{\text{new}} = \emptyset$ }

If ($X_{\text{now}} \neq \emptyset$) { $P_{\text{now}} = P_{\text{all}}$

Receive ($\hat{B}_j^{(q+1)}$) from all Sink nodes.

If ($\hat{B}_j^{(q+1)} = 0$) $P_{\text{now}} = P_{\text{now}} - K_j$, $\forall K_j$, $j \in J$

Step 2: Determine $\alpha_j^{(q+1)}$ & $T^{(q+1)}$:

$\alpha_j^{(q+1)} = 1/(w_j \sum_{x=1}^{M} \frac{1}{w_x})$, $\forall K_j \in P_{\text{now}}$, $j \in J$

$L = \sum_{i=1}^{N} L_i$, $\forall S_i \in X_{\text{now}}$, $i \in I$

$Y = \min\{\hat{B}_j^{(q+1)}/(\alpha_j^{(q+1)} L)$, $\forall K_j \in P_{\text{now}}$, $j \in J\}$

If ($Y > 1$) $\{Y = 1\}$

$T^{(q+1)} = Y \alpha_j^{(q+1)} L \; w_j \; T_{\text{cp}}$, *for any* K_j , $j \in J$

Broadcast the schedule information (Y, $\alpha_j^{(q+1)}$, L, $(T + T^{(q+1)})$, $(L_i, \forall S_i \in X_{\text{now}}, i \in I)$) to all the Sink nodes.

Step 3: Update the amount of load remaining to be processed:

Wait till $(T + T^{(q)})$ & Receive (L_j) from all the Sink nodes.

$q = q + 1$

$L_i = L_i \cdot (1 - Y + ((\sum_{j=1}^{M} L_j)/L))$, $\forall S_i \in X_{\text{now}}$, $i \in I$

{ If ($L_i = 0$) $\{X_{\text{now}} = X_{\text{now}} - \{S_i\}\}$}, $\forall S_i \in X_{\text{now}}$, $i \in I$ }

Go to **Step 1**.

Figure 4.9: Pseudo code describing the workings of the Adaptive IBS algorithm at the coordinator node C_s.

Initial state:

$q = 0$, $T^{(0)} = 0$, $p = 0.95$

Step 1: Buffer availability estimation for next iteration:

$\hat{B}_j^{(q+1)} = ((\sum_{k=0}^{(s-1)} ((s-k) \cdot B_j^{(q-k)})) / (\sum_{k=0}^{(s-1)} k)) * p$

Send $(\hat{B}_j^{(q+1)})$ to the Coordinator node.

Step 2: Wait till the previous iteration is completed:

Wait till $(T + T^{(q)})$.

Step 3: Compute Load amounts to be processed:

Receive the schedule information (Y, $\alpha_j^{(q+1)}$, L, $(T + T^{(q+1)})$, $(L_i, \forall S_i \in X_{\text{now}}, i \in I)$) from the Coordinator node.

$q = q + 1$

If $(\hat{B}_j^{(q)} \neq 0)$ {

If $((Y\alpha_j^{(q)}L) > B_j^{(q)})$ { $B_j^{(q)} = 0$

$L_j = (Y\alpha_j^{(q)}L) - B_j^{(q)}$

$\alpha_{i,j}^{(q)} = L_i \cdot \frac{B_j^{(q)}}{L}$, $\forall S_i \in X_{\text{now}}$, $i \in I$ }

Else { $L_j = 0$

$B_j^{(q)} = B_j^{(q)} - (Y\alpha_j^{(q)}L)$

$\alpha_{i,j}^{(q)} = Y\alpha_j^{(q)}L_i$, $\forall S_i \in X_{\text{now}}$, $i \in I$ }

Send (L_j) to the Coordinator node.

Step 4: Schedule the loads from Source Nodes:

Request, receive and process the load fractions $(\alpha_{i,j}^{(q)})$ from the Source Nodes $S_i \in X_{\text{now}}$.

}

Go to **Step 1**.

Figure 4.10: Pseudo code describing the workings of the Adaptive IBS algorithm at the sink nodes.

4.2.1 Buffer Estimation Strategy

We propose a distributed buffer estimation strategy based on weighted average calculations of buffer availability in the previous "s" iterations. The weights for computing the estimates are based on the iteration indices until the current iteration. We refer to this estimator as *Iteration Index based Buffer estimator* (IIB). Our IIB algorithm shall be executed at all sink nodes. A sink node, after estimating the buffer space to render in the next iteration, shall communicate it to the coordinator node so that it could determine the scheduling parameters required for the sink nodes.

For estimating the buffer availability at a sink, each sink K_j needs to keep track of the actual buffer sizes B_j from its previous "s" iterations. In an iteration q, each sink node shall estimate the buffer size that will be available for the next iteration $(q+1)$ as

$$\hat{B}_j^{(q+1)} = \left(\frac{\sum_{k=0}^{(s-1)} ((s-k) \cdot B_j^{(q-k)})}{\sum_{k=0}^{(s-1)} k} \right) \cdot p \tag{4.7}$$

and declare it to the coordinator node. In (4.7), p is the probability that the estimated buffer size will be available at a sink at the next iteration. The value of p can be chosen based on the confidence level of the buffer estimator. For practical purposes we shall assume that p equals 0.95. This guarantees that the expected buffer sizes will be available at the sinks, with a confidence level of 95%, for the next iteration.

Example 4.3 clarifies the working principle of the Adaptive IBS algorithm. The sink

speed ($\frac{1}{w_j}$) parameters for this example are derived from the STAR experiments conducted at BNL [51]. The time at which the buffer capacity at the sink nodes vary are chosen so as to illustrate the finer details of the algorithm.

Example 4.3**:**

Let us suppose that there are three sources with loads to be processed and there are four sinks that can process these loads. Let the speed parameter of sinks be $w_1 = 1.11 \times 10^{-9}$, $w_2 = 6.25 \times 10^{-10}$, $w_3 = 5.00 \times 10^{-10}$ and $w_4 = 3.57 \times 10^{-10}$, respectively. Let $T_{\text{cp}} = 6.52 \times 10^{12}$sec/load. Let the buffer capacities at sinks at time $t = 0$ seconds be $B_1 = 6$, $B_2 = 5$, $B_3 = 0$, and $B_4 = 2$; at time $t = 4.655 \times 10^3$ seconds be $B_1 = 4$, $B_2 = 3$, $B_3 = 1$, and $B_4 = 1$; at time $t = 8.607 \times 10^3$ seconds be $B_1 = 2$, $B_2 = 0$, $B_3 = 2$, and $B_4 = 1$; and at time $t = 10.67 \times 10^3$ seconds be $B_1 = 1$, $B_2 = 1$, $B_3 = 3$, and $B_4 = 1$ units respectively. These values are generated randomly using a uniform probability distribution in the range $[0, 7]$. We let the three sources to have loads $L_1 = 5$, $L_2 = 2$ and $L_3 = 3$ unit loads, respectively. Let loads L_1 and L_2 arrive at $t = 0$ seconds, and load L_3 arrive at $t = 5 \times 10^3$ seconds. The sink and source node parameters are summarized in the Table 4.5.

Using the algorithm in Fig. 4.9 and 4.10, we have the values for $\alpha_{i,j}^{(q)}$ as shown in Tables 4.6 and 4.7. The estimated and actual values for the load fractions to be processed and the buffer availabilities at the sink nodes at various iterations are shown in Table 4.6 and Fig. 4.11. The unutilized buffer space in all the iterations

Table 4.5: Sink and Source node parameters for Example 4.3.

Sink nodes	**Parameter**	
	Inverse of computing speed (w_j)	**Buffer capacity (B_j)**
Sink node 1 (K_1)	1.11×10^{-9}	6 [at 0 sec]
		4 [at 4.655×10^3 sec]
		2 [at 8.607×10^3 sec]
		1 [at 10.67×10^3 sec]
Sink node 2 (K_2)	6.25×10^{-10}	5 [at 0 sec]
		3 [at 4.655×10^3 sec]
		0 [at 8.607×10^3 sec]
		1 [at 10.67×10^3 sec]
Sink node 3 (K_3)	5.00×10^{-10}	0 [at 0 sec]
		1 [at 4.655×10^3 sec]
		2 [at 8.607×10^3 sec]
		3 [at 10.67×10^3 sec]
Sink node 4 (K_4)	3.57×10^{-10}	2 [at 0 sec]
		1 [at 4.655×10^3 sec]
		1 [at 8.607×10^3 sec]
		1 [at 10.67×10^3 sec]
Source nodes	**Parameter**	
	Load Size (L_i)	**Load arrival time**
Source node 1 (S_1)	5	0 sec
Source node 2 (S_2)	2	0 sec
Source node 3 (S_3)	3	5×10^3 sec

Table 4.6: Buffer utilization values for Example 4.3.

$\mathbf{q = 1}$	$\sum \hat{\alpha}_{\mathbf{i,j}}^{(\mathbf{1})}$	$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{1})}$	$\hat{\mathbf{B}}_{\mathbf{j}}^{(\mathbf{1})}$	$\mathbf{B}_{\mathbf{j}}^{(\mathbf{1})}$
K_1	0.643	0.643	6.000	5.357
K_2	1.143	1.143	5.000	3.857
K_3	0.000	0.000	0.000	0.000
K_4	2.000	2.000	2.000	0.000
$\mathbf{q = 2}$	$\sum \hat{\alpha}_{\mathbf{i,j}}^{(\mathbf{2})}$	$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{2})}$	$\hat{\mathbf{B}}_{\mathbf{j}}^{(\mathbf{2})}$	$\mathbf{B}_{\mathbf{j}}^{(\mathbf{2})}$
K_1	0.546	0.546	5.700	3.454
K_2	0.970	0.970	4.750	2.030
K_3	0.000	0.000	0.000	1.000
K_4	1.698	1.000	1.900	0.000
$\mathbf{q = 3}$	$\sum \hat{\alpha}_{\mathbf{i,j}}^{(\mathbf{3})}$	$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{3})}$	$\hat{\mathbf{B}}_{\mathbf{j}}^{(\mathbf{3})}$	$\mathbf{B}_{\mathbf{j}}^{(\mathbf{3})}$
K_1	0.284	0.284	4.433	1.716
K_2	0.506	0.000	3.483	0.000
K_3	0.633	0.633	0.633	1.367
K_4	0.886	0.886	1.267	0.114
$\mathbf{q = 4}$	$\sum \hat{\alpha}_{\mathbf{i,j}}^{(\mathbf{4})}$	$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{4})}$	$\hat{\mathbf{B}}_{\mathbf{j}}^{(\mathbf{4})}$	$\mathbf{B}_{\mathbf{j}}^{(\mathbf{4})}$
K_1	0.235	0.235	3.167	0.765
K_2	0.415	0.415	1.742	0.585
K_3	0.518	0.518	1.267	2.482
K_4	0.727	0.727	1.108	0.273

Table 4.7: Load fraction values for Example 4.3.

$\mathbf{q = 1}$	$\mathbf{S_1}$	$\mathbf{S_2}$		$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{1})}$
K_1	0.459	0.184		0.643
K_2	0.817	0.326		1.143
K_3	0.000	0.000		0.000
K_4	1.429	0.571		2.000
$\mathbf{q = 2}$	$\mathbf{S_1}$	$\mathbf{S_2}$		$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{2})}$
K_1	0.390	0.156		0.546
K_2	0.693	0.277		0.970
K_3	0.000	0.000		0.000
K_4	0.714	0.286		1.000
$\mathbf{q = 3}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$	$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{3})}$
K_1	0.038	0.015	0.231	0.284
K_2	0.000	0.000	0.000	0.000
K_3	0.085	0.034	0.514	0.633
K_4	0.119	0.048	0.719	0.886
$\mathbf{q = 4}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$	$\sum \alpha_{\mathbf{i,j}}^{(\mathbf{4})}$
K_1	0.032	0.013	0.190	0.235
K_2	0.056	0.022	0.337	0.415
K_3	0.070	0.028	0.420	0.518
K_4	0.098	0.040	0.589	0.727

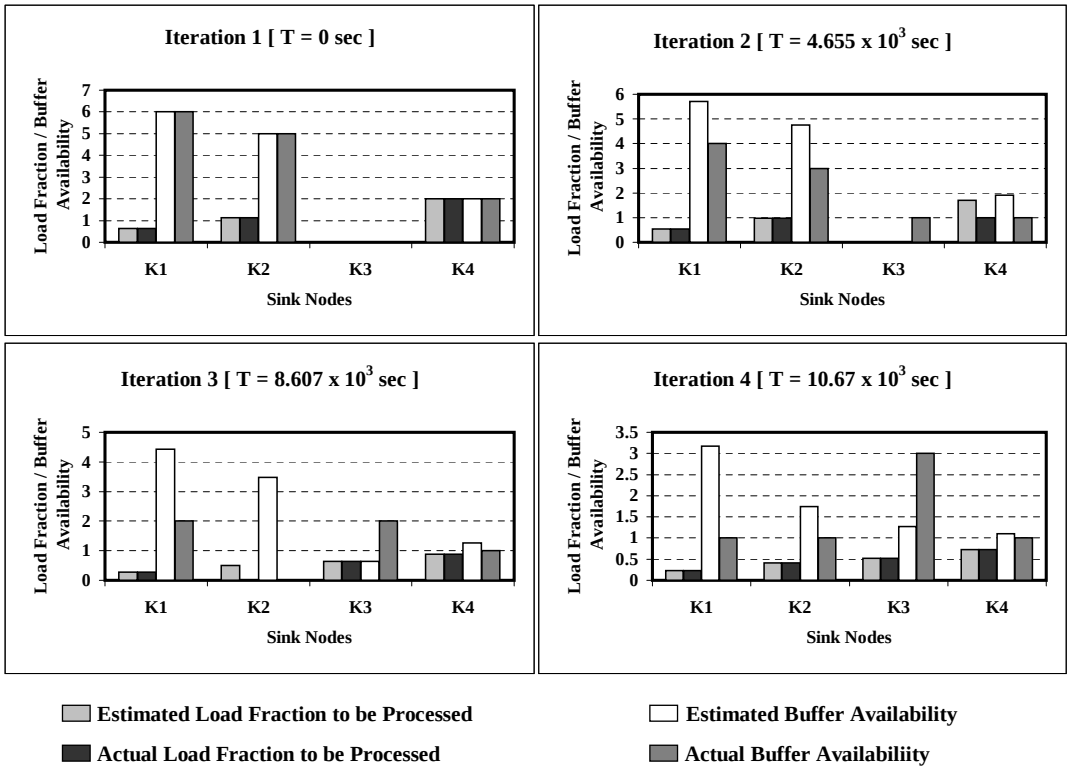

Figure 4.11: The estimated and actual values for the load fractions to be processed and the buffer availabilities at the sink nodes at various iterations.

is given in the last column of Table 4.6. From, these results, we observe that buffer of K_4 is fully utilized in iterations 1 and 2, whereas the available buffer at K_3 is not at all utilized in iteration 2 (because estimated buffer size is 0 for that iteration). For iteration 3, buffer of K_3 is estimated to be less than the actual value and hence buffers of all the available sinks are under utilized in that iteration. At the final iteration, the remaining load is insufficient to completely fill up the buffer at any of the sinks. The distribution suggested by the values $\alpha_{i,j}$ in the Table 4.7 are used by the sinks. Iteration 1 to 4 are scheduled at time $t = 0,\ 4.655 \times 10^3,\ 8.607 \times 10^3,$ and 10.67×10^3 seconds, respectively. The total processing time for processing all the three loads is $t = 12.36 \times 10^3$ seconds. The estimated and actual load fractions from the source nodes to be processed and the total load in the system at various iterations are as shown in Fig. 4.12. From this example, it is seen that, because of the buffer space variations at the sink K_3, the processing for the sources S_1 and S_2 could not be completed in the iteration 2. And, because of the arrival of new source S_3, the processing time for the other sources in the system (S_1 and S_2) are stretched to $t = 12.36 \times 10^3$ seconds.

The impact of IIB is as follows. In Table 4.6 the estimated as well as the actual loads requested by the sinks are presented. Further we also project the estimated buffer values. In iteration 1, the estimated and the actual loads being same, the buffer rendered is adequate to handle the estimated load. However, in iteration 2, we observe that at K_4, the estimated load being more than the actual buffer rendered, the actual load that is to be requested is tailored to adapt to the available

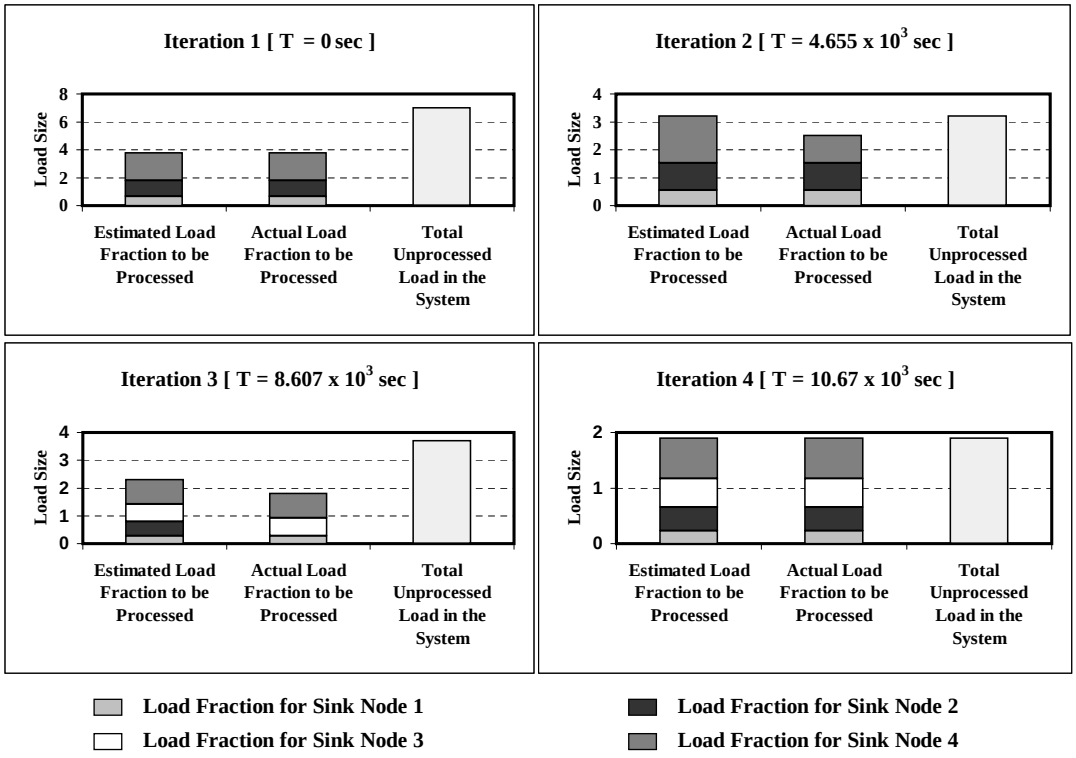

Figure 4.12: Performance of Adaptive IBS algorithm.

space. It may also be observed that in iteration 2, the estimated buffers take into account the actual buffers rendered in the past iteration. This will be cumulatively done in each iteration, which is indeed the essence of our design. Further, in iteration 2, the actual buffer available at K_3 is unutilized, as the estimated value is 0. This is a natural behavior that is captured in our design. Another important observation comes from the fact that in iteration 2, if the estimated load sizes have been requested by all the sinks then the processing for sources S_1 and S_2 could have been completed in this iteration itself. However since K_4 could not accommodate the estimated load, S_1 and S_2 are forced to be considered for scheduling in the future iterations as well.

Also, note that although S_3 becomes available for processing after iteration 2 starts, it is considered for processing in iteration 3 onwards. Note that in iteration 3, the estimated buffer at K_3 is observed to be less than the actual buffer available. Thus, the scheduler considers a load based on a minimum of the actual or estimated buffer space. In this case, it turns out to be the estimated buffer value. Now, when the estimated total load to be processed is less than or equal to the available buffer spaces, then all the loads could be scheduled and processed at this iteration itself. This happens at the final iteration.

The proposed IIB strategy works as long as the buffer variations are not drastic. Further, if new loads arrive to the system before the loads being processed are completed, then the processing of existing loads will be stretched. Thus, when loads to be processed are not time critical this strategy is highly recommended,

since it adapts to buffer variations at sinks as well.

Chapter 5

Scheduling Strategies for Time Critical Loads

In all the algorithms described in the last chapter, it was shown that as and when new sources are added to the system, the processing time of all the other sources already in the system are stretched or extended. Hence, they become a natural choice for cluster systems that process loads that are not time and/or mission critical. However, for real-time processing of loads, which are indeed bound to guarantee the completion of processing on or before a specified deadline, otherwise referred to as deadline requirements hereafter, the algorithms may be forced to consider only a limited set of loads for processing.

In this chapter, we shall propose our resource aware dynamic incremental scheduling strategies for cluster systems having time and mission critical loads and sink nodes

having finite buffer capacity constraints.

5.1 Resource Aware Dynamic Incremental Scheduling Strategies

We now describe our Resource Aware Dynamic Incremental Scheduling (RADIS) strategies for scheduling within cluster systems. Similar to the Dynamic and Adaptive IBS algorithms presented in the last chapter, here too we assume that the coordinator node C_s computes the parameters required by the sink nodes to determine a schedule satisfying the resource constraints. We consider three different scheduling strategies, namely *Non-interleaved Scheduling Scheme*, *Earliest Deadlines First Scheme* (EDF) [61], and *Progressive Scheduling Scheme* for dynamic environments, depending on how the set of loads are to be processed. All our strategies work in an incremental fashion, consuming several iterations for scheduling the loads. Each iteration refers to a time period in which a set of sinks are to be scheduled for processing the loads by the node C_s.

Below we will describe the workings of our scheduler in the coordinator and sink nodes in a systematic fashion. Flow-charts shown in Fig. 5.1, 5.2, and 5.3, and the pseudo-codes in Fig. 5.6, 5.7 and 5.8 describe the workings of the coordinator node, the admission control procedure and the sink nodes respectively.

In our strategies, in every iteration, after the admissibility testing for the newly

arrived sources, the scheduler at the C_s first determines the loads to be scheduled and sinks that will participate. Based on the estimated buffer availabilities at the sink nodes, C_s computes an estimate of the amount of load to be scheduled at a sink node and the finish time for the next iteration. It then broadcasts this schedule information to all the sink nodes. A sink node upon receiving this information will wait for the current iteration to be completed and determines the actual buffer availability for the next iteration. Based on the estimate received from the C_s and its actual buffer availability, it computes the amount of load it can process in the next iteration and requests that load from the respective sources. It also communicates the difference between the estimated and the actual amount of load to the C_s.

We shall first consider handling the time-varying buffer availabilities at the sink nodes and then the dynamic arrival of loads. As described in Chapter 4, for dynamic environments, a feasible schedule may not exist unless the sink nodes allow their available buffers to be reused after a given load is processed. Hence, here again, we assume that the sink nodes allow their available buffer spaces to be reused after processing is completed in an iteration so as to enable scheduling more amounts of loads and incrementally process the loads. As in the Adaptive IBS algorithm presented in Chapter 4, our RADIS strategies also take into account that the buffer space variations at sinks may not be known a priori.

In our strategies, the sinks estimate the amount of buffer space that they could offer for scheduling in the next iteration as described in Chapter 4, and communicate it

to the coordinator node C_s. With this information, C_s determines the participating sink nodes for the next iteration, computes the required parameters to schedule the loads in an incremental fashion and communicates them to all the sink nodes.

The sink nodes receive the information from C_s and waits till the processing is completed by all the sink nodes. Then, if at a sink, the actual buffer availability is not sufficient to accommodate the estimated amount of load, then that sink node computes the load fractions to be requested from the source nodes as given in (4.6).

If the buffer availability at a sink node is more or enough to accommodate the estimated load fraction, the sink node computes the load fraction to be requested from the source node S_i in the iteration q as

$$\alpha_{i,j}^{(q)} = Y \ \alpha_j^{(q)} \ L_i \tag{5.1}$$

The sink nodes communicate to C_s the difference between the amount of load that they estimated to process and the actual amount of load that they are processing. All the sinks request from sources these load fractions and process them. Following our model described in Chapter 2, all the sinks start computing the load fractions as they start receiving them from the sources.

The coordinator node C_s receives the information on the difference between the amount of load estimated to be processed and the actual amount of load processed

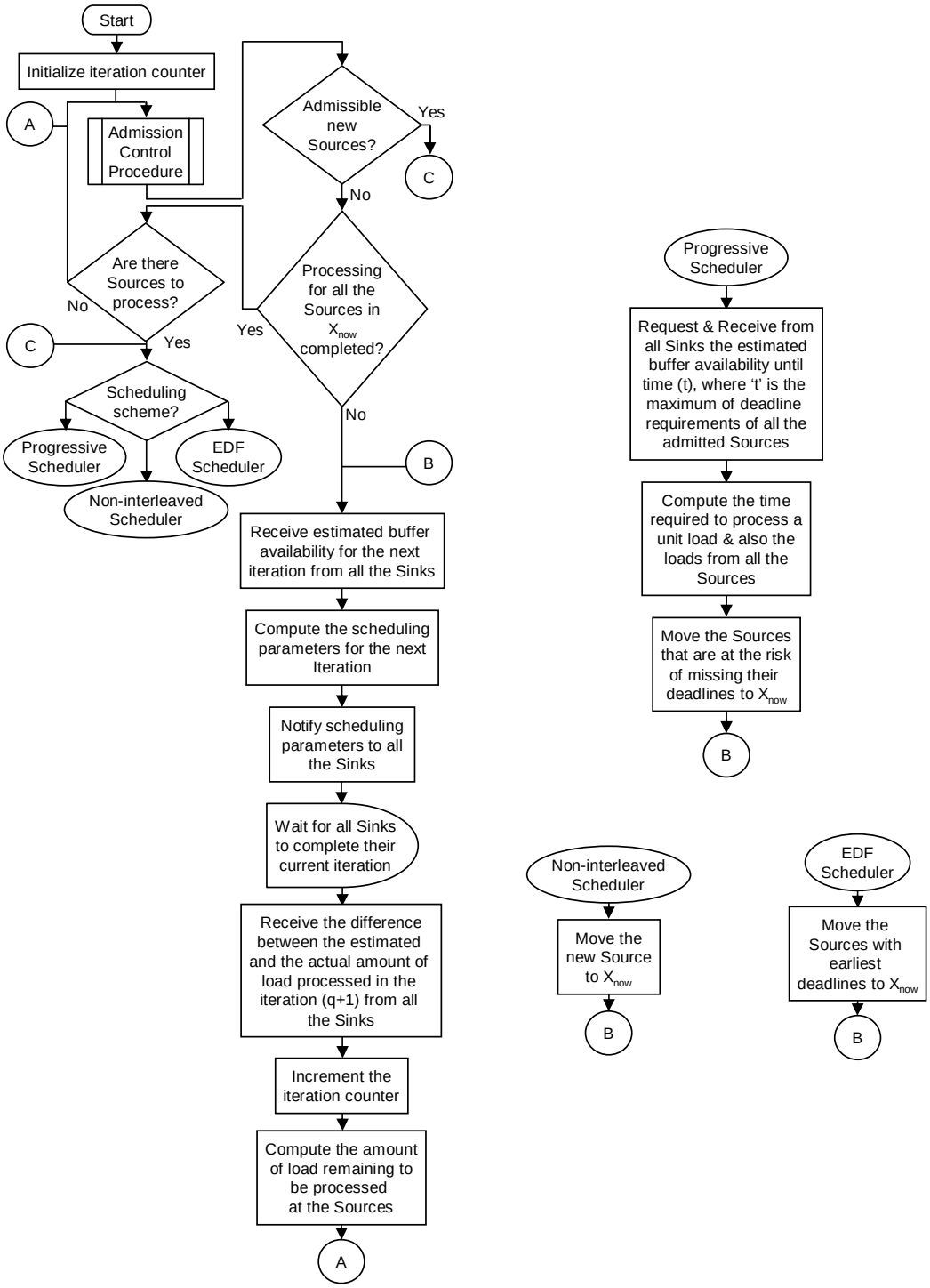

Figure 5.1: Flowchart for the workings of the RADIS scheduler at the coordinator node C_s.

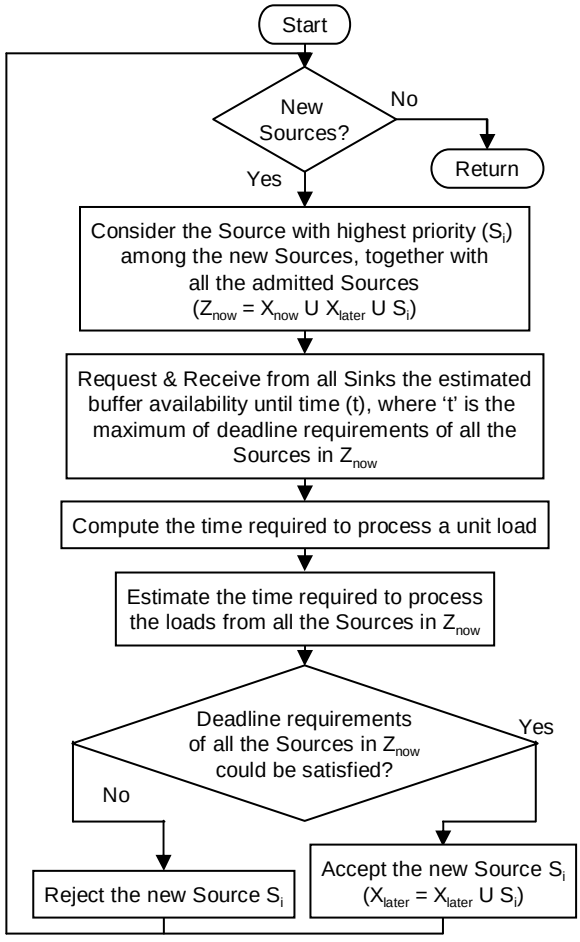

Figure 5.2: Flowchart for the workings of the admission control procedure at the coordinator node C_s.

at the participating sink nodes, computes the remaining amount of load in the system and waits till the processing is completed by all the sink nodes for that iteration. It may be noted that, since, buffer availability is a function of time, the guarantees given to the sources (to complete processing within their deadlines) may be met only when the buffer estimation strategy utilized is conservative.

In RADIS, the total load processed in the q^{th} iteration is given by

$$\sum_{j=1}^{M} \alpha_j^{(q)} \;=\; \sum_{i=1}^{N}\sum_{j=1}^{M} \alpha_{i,j}^{(q)} \tag{5.2}$$

Hence, the time taken to process a unit load in the q^{th} iteration is given by

$$T_{\text{ul}} \;=\; \frac{T^{(q)}}{\sum_{i=1}^{N}\sum_{j=1}^{M} \alpha_{i,j}^{(q)}} \tag{5.3}$$

RADIS attempts to completely fill at least one of the sink's buffers in every iteration, depending on the processing speed and the size of the buffer available at the sinks. Since, in our strategy all the sinks are forced to stop processing at the same time, the product of buffer utilization and the inverse of computing speed for each sink node will be the same. From this the maximum buffer utilization, or in other words, the total load that could be processed at each sink in an iteration could be derived as,

$$\sum_{i=1}^{N} \alpha_{i,j} = \frac{B_{i^*} w_{i^*}}{w_j}, \text{ where } i^* = \text{ argmin } \{\frac{B_j}{\alpha_j}\} \tag{5.4}$$

In (5.4), the "argmin" term identifies a sink whose buffer is completely filled. The

buffer utilization at other sink nodes, and hence, the total amount of buffer utilized for optimal processing by the system, or in other words, the total load that could be processed by the system in an iteration is computed as,

$$\sum_{i=1}^{N}\sum_{j=1}^{M}\alpha_{i,j} = \sum_{j=1}^{M}\frac{B_{i^*}w_{i^*}}{w_j} \tag{5.5}$$

Substituting (4.1), (5.4), and (5.5) into (5.3), the time taken to process a unit load is computed as

$$T_{\text{ul}} \;=\; \frac{T_{\text{cp}}}{\sum_{j=1}^{M}\;\{\;\frac{1}{w_j}\;\}} \tag{5.6}$$

Hence, the estimated time taken to process the loads in the system is given by

$$\hat{T} \;=\; T_{\text{ul}} \cdot \sum_{i=1}^{N} L_i \tag{5.7}$$

Thus, the time taken to process the loads from the sources that are being considered are estimated in RADIS strategies.

Now, we shall discuss on how our strategies consider the dynamic arrival of loads. In RADIS, in every iteration, if there are new sources, C_s considers them in their priority order. It then requests the sink nodes to estimate their buffer availabilities until the farthest deadline requirement time of all the sources that were accepted earlier and also the new source that is being considered. The sinks estimate their buffer availability by calculating the time average of their historical data as given

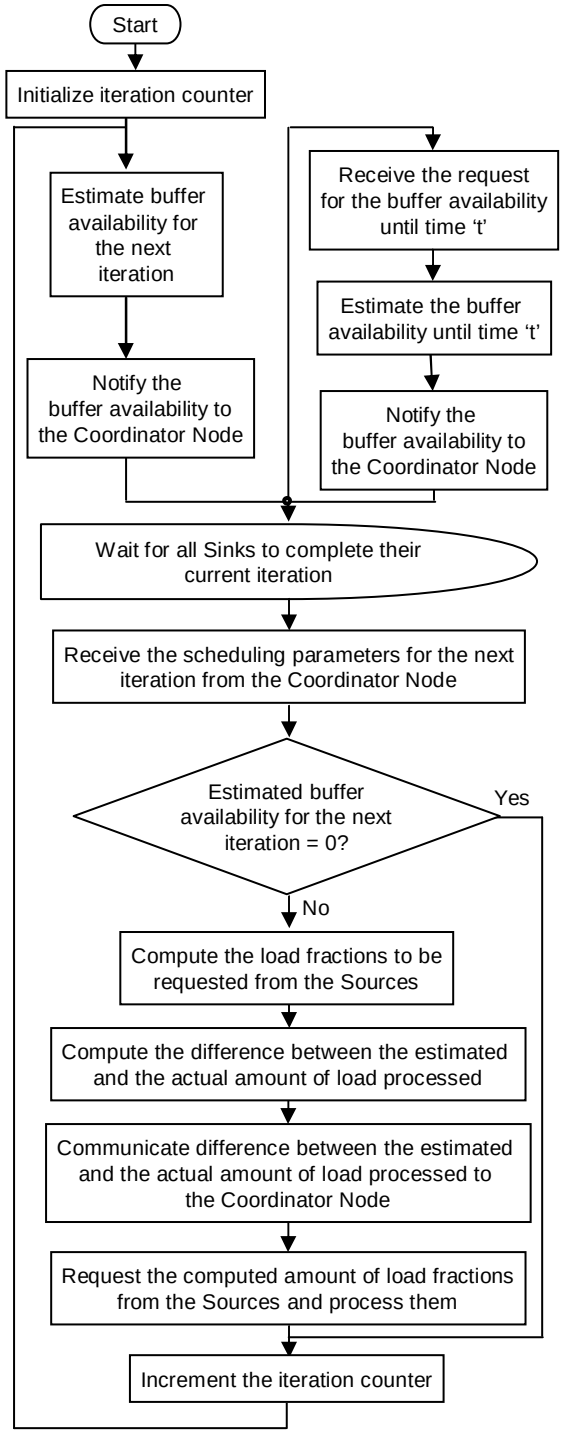

Figure 5.3: Flowchart for the workings of the RADIS scheduler at the sink nodes.

by

$$\hat{B}_j^t \;=\; \frac{1}{t_{\text{req}}} \int_{t=(T-t_{\text{req}})}^{t=T} B_j(t)\; dt \tag{5.8}$$

where $t_{\text{req}} \;=\; (\max\{T_{\text{d}_i}\} \;-\; T)$, $(T - t_{\text{req}}) \geq 0$ and $\max\{T_{\text{d}_i}\}$ is the farthest deadline requirement time of all the sources, and communicates it to C_s. Then, C_s decides on the set of sources to be scheduled in that iteration, based on the deadline requirements and estimated buffer availabilities.

The estimation requests could be further minimized, if C_s requests the sink nodes to estimate the buffer variations considering the deadline requirements of all the new sources in every iteration. This approach has an inherent advantage of minimizing the communication required for estimating the buffer for admissibility testing, especially when the source arrival rates are higher.

In RADIS, the set of loads that are scheduled in an iteration and the admission criteria for the sources vary for different scheduling schemes and are discussed in Section 5.1.1, 5.1.2, and 5.1.3 for Non-interleaved, EDF, and Progressive scheduling strategies respectively. The new set of loads and the unprocessed loads from the existing sources are considered together for scheduling at the start of every iteration. This process is continued until all the loads are processed.

The proposed admissibility criterion together with the conservative buffer estimation strategy guarantees that deadlines for all the loads that are accepted will always be satisfied. It may be noted that in RADIS the priorities of the sources are used only to resolve conflicts that may arise while admitting multiple sources.

If multiple sources have identical priorities set, schemes such as FIFO or other possible heuristics can be adopted to resolve the conflict.

A simulation study presented in the Section 5.3 clarifies the workings of RADIS in detail.

5.1.1 Non-interleaved Scheduling Strategy

In the Non-interleaved scheduling strategy,

- in every iteration all the sources that were admitted into the system are scheduled for processing, and,
- during admissibility testing, the algorithm admits the new source only if the deadline requirement of all the sources that were admitted earlier and also that of the new source could be satisfied when they all are scheduled together in every iterations, or else the new source is not admitted into the system.

5.1.2 Earliest Deadline First Scheduling Strategy

In the EDF strategy,

- in every iteration among the sources that are admitted into the system, the sources with earliest deadlines are considered for processing, and
- during admissibility testing, the algorithm checks the deadline requirement

of all the sources that were admitted earlier and also the new source against the processing time required to process them, considering the sources with earliest deadline first. The process is repeated until the deadline requirements of all of the sources are found to be satisfied, in which case the new source shall be admitted, or the deadline requirement of some of the sources is violated, in which case the new source shall not be admitted into the system.

5.1.3 Progressive Scheduling Strategy

In the Progressive scheduling strategy, in every iteration the loads from the sources that are at the risk of missing their deadlines are processed. Here in, there are three possibilities and the actions taken under such situations are as follows:

(a) *Deadline requirements for all sources considered are later than $\hat{T}$:* Since, the deadline requirements of all the sources in the system could be satisfied, the set of sources that were considered previously are scheduled.

(b) *Deadline requirements for all sources considered are earlier than $\hat{T}$:* Here, there is a chance that the deadline requirements of a set of sources that were considered previously may not be satisfied. Hence, those sources are scheduled immediately.

(c) *Deadline requirements for some of the sources are earlier and some are later than $\hat{T}$:* Since the deadline requirements of some of the sources are earlier than $\hat{T}$, we reiterate considering only those sources.

Similarly, there exists three possibilities during the admissibility testing for a new source, and different actions are taken for them as follows:

(a) *Deadline requirements for all sources considered are later than $\hat{T}$:* Since, the deadline requirements of all the sources could be satisfied, the new source that is considered can be accepted.

(b) *Deadline requirements for all sources considered are earlier than $\hat{T}$:* Here, the new source that is considered cannot be accepted. Note that the priority of a source could depend on its processing requirements.

(c) *Deadline requirements for some of the sources are earlier and some are later than $\hat{T}$:* Under this condition, we reiterate considering only those sources whose deadlines are earlier than $\hat{T}$.

This interleaving scheme works in a style contrary to most of the conventional schedulers that uses priority as the criteria for scheduling the loads. In this scheme, some of the sources that are processed in an iteration could be suspended and some other sources could be scheduled in the following iteration, so as to satisfy the deadline requirements of the admitted sources.

Example 5.1 and 5.2 clarifies the working principle of the Progressive scheduling strategy in time-invariant and predictable time-varying buffer environment respectively. The sink speed ($\frac{1}{w_j}$) parameters for this example are derived from the STAR experiments conducted at BNL [51].

Example 5.1**:**

Let us consider a four sink node system with the speed parameters $w_1 = 1.11 \times 10^{-9}$, $w_2 = 6.25 \times 10^{-10}$, $w_3 = 5.00 \times 10^{-10}$ and $w_4 = 3.57 \times 10^{-10}$ respectively. Let $T_{\text{cp}} = 6.52 \times 10^{12}$sec/load. Let the buffer capacities at sinks be $B_1 = 6$, $B_2 = 5$, $B_3 = 2$, and $B_4 = 3$ respectively. We let the nine sources to have loads $L_1 = 5$, $L_2 = 3$, $L_3 = 9$, $L_4 = 6$, $L_5 = 4$, $L_6 = 3$, $L_7 = 3$, $L_8 = 5$ and $L_9 = 1$ units. The values for loads and buffer capacities are generated randomly using a uniform probability distribution in the range $[0, 9]$. Let load arrival times in the units of $\times 10^3$ seconds be L_1 to $L_3 = 0$, $L_4 = 2$, $L_5 = 4$, $L_6 = 9$, $L_7 = 10$, $L_8 = 12$ and $L_9 = 13$ respectively. Let the firm deadlines demanded by the sources S_1 to S_9 in the units of $\times 10^3$ seconds be 5, 3, 20, 8, 20, 15, 15, 20 and 20 respectively. Let the priorities of the sources S_1 to S_9 be 5, 2, 0, 4, 3, 2, 1, 3 and 2 respectively, where 0 is the lowest and 5 is the highest priority. The sink and source node parameters are summarized in the Table 5.1.

The results are summarized in Table 5.2. K_4 is the fastest sink in the system. But, our scheduler tries to use the buffer of K_3 to the fullest, since it considers the combined effect of the speed and the buffer availability at the sinks while determining it. From the table, we observe that the buffer of K_3 is fully consumed only at the iteration 2, when the total load in the system is more than the optimal buffer size of the system. At the other iterations, the load considered for processing is insufficient to completely fill up the buffer of any of the sinks. The values of $\alpha_{i,j}$

Table 5.1: Sink and Source node parameters for Example 5.1.

Sink nodes	**Parameter**	
	Inverse of computing speed (w_j)	**Buffer capacity (B_j)**
Sink node 1 (K_1)	1.11×10^{-9}	6
Sink node 2 (K_2)	6.25×10^{-10}	5
Sink node 3 (K_3)	5.00×10^{-10}	2
Sink node 4 (K_4)	3.57×10^{-10}	3

Source nodes	**Parameter**			
	Load Size (L_i)	**Load arrival time**	**Deadline**	**Priority**
Source node 1 (S_1)	5	0 sec	5×10^3 sec	5
Source node 2 (S_2)	3	0 sec	3×10^3 sec	2
Source node 3 (S_3)	9	0 sec	20×10^3 sec	0
Source node 4 (S_4)	6	2×10^3 sec	8×10^3 sec	4
Source node 5 (S_5)	4	4×10^3 sec	20×10^3 sec	3
Source node 6 (S_6)	3	9×10^3 sec	15×10^3 sec	2
Source node 7 (S_7)	3	10×10^3 sec	15×10^3 sec	1
Source node 8 (S_8)	5	12×10^3 sec	20×10^3 sec	3
Source node 9 (S_9)	1	13×10^3 sec	20×10^3 sec	2

Table 5.2: Load fraction and buffer utilization values for Example 5.1.

$\mathbf{q=1}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$		$\sum \alpha_{i,j}^{(1)}$	$\mathbf{B_j^{(1)}}$
K_1	0.6170	0.00	0.00		0.6170	5.3830
K_2	1.0955	0.00	0.00		1.0955	3.9045
K_3	1.3695	0.00	0.00		1.3695	0.6305
K_4	1.9180	0.00	0.00		1.9180	1.0820
$\mathbf{q=2}$	$\mathbf{S_3}$	$\mathbf{S_4}$	$\mathbf{S_5}$		$\sum \alpha_{i,j}^{(2)}$	$\mathbf{B_j^{(2)}}$
K_1	0.62	0.00	0.28		0.90	5.10
K_2	1.11	0.00	0.49		1.60	3.40
K_3	1.38	0.00	0.62		2.00	0.00
K_4	1.94	0.00	0.86		2.80	0.20
$\mathbf{q=3}$	$\mathbf{S_3}$	$\mathbf{S_5}$	$\mathbf{S_6}$	$\mathbf{S_7}$	$\sum \alpha_{i,j}^{(3)}$	$\mathbf{B_j^{(3)}}$
K_1	0.00	0.00	0.3702	0.00	0.3702	5.6298
K_2	0.00	0.00	0.6573	0.00	0.6573	4.3427
K_3	0.00	0.00	0.8217	0.00	0.8217	1.1783
K_4	0.00	0.00	1.1508	0.00	1.1508	1.8492
$\mathbf{q=4}$	$\mathbf{S_3}$	$\mathbf{S_5}$	$\mathbf{S_8}$	$\mathbf{S_9}$	$\sum \alpha_{i,j}^{(4)}$	$\mathbf{B_j^{(4)}}$
K_1	0.487	0.216	0.00	0.123	0.826	5.174
K_2	0.865	0.383	0.00	0.219	1.467	3.533
K_3	1.082	0.479	0.00	0.274	1.835	0.165
K_4	1.516	0.672	0.00	0.384	2.572	0.428

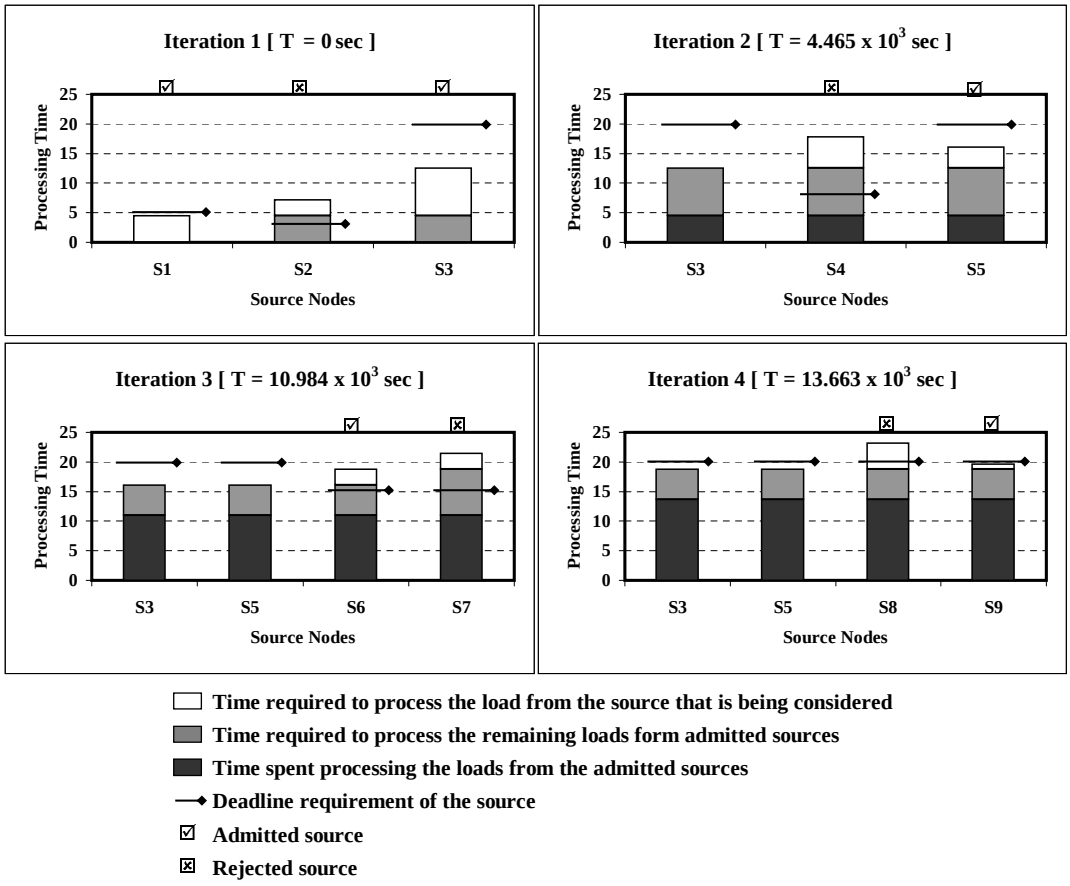

Figure 5.4: Performance of Progressive scheduling strategy in time-invariant buffer environment.

for the four iterations are computed at time instants $t = 0,\ 4.465 \times 10^3,\ 10.984 \times 10^3$ and 13.663×10^3 seconds respectively. The total processing time is $t = 19.646 \times 10^3$ seconds. The processing time and deadline requirements of the source nodes at various iterations are as shown in Fig. 5.4 and are given in $\times 10^3$ seconds.

Example 5.2**:**

Let us consider a four sink node system with the speed parameters $w_1 = 1.11 \times 10^{-9}$, $w_2 = 6.25 \times 10^{-10}$, $w_3 = 5.00 \times 10^{-10}$ and $w_4 = 3.57 \times 10^{-10}$ respectively. Let $T_{\text{cp}} = 6.52 \times 10^{12}$sec/load. Let the estimated buffer capacities at sinks at time $t = 0$ seconds be $B_1 = 6$, $B_2 = 2$, $B_3 = 0$, and $B_4 = 3$; at time $t = 10 \times 10^3$ seconds be $B_1 = 2$, $B_2 = 3$, $B_3 = 2$, and $B_4 = 3$; and at time $t = 20 \times 10^3$ seconds be $B_1 = 0$, $B_2 = 1$, $B_3 = 1$, and $B_4 = 1$, respectively. We let the 9 sources to have loads $L_1 = 3$, $L_2 = 4$, $L_3 = 3$, $L_4 = 2$, $L_5 = 5$, $L_6 = 2$, $L_7 = 2$, $L_8 = 5$ and $L_9 = 1$ units. The values for loads and buffer capacities are also generated randomly using a uniform probability distribution in the range $[0, 7]$. It shall be noted that a value zero for buffer capacity corresponds to a situation wherein a sink will not participate, thus reflecting a real-life situation. Let load arrival times in the units of $\times 10^3$ seconds be L_1 to $L_3 = 0$, $L_4 = 2$, $L_5 = 3$, $L_6 = 9$, $L_7 = 8$ and L_8 and $L_9 = 11$ respectively. Let the firm deadlines demanded by the sources S_1 to S_9 in the units of $\times 10^3$ seconds be 6, 8, 16, 5, 16, 14, 15, 13 and 16 respectively. Let the priorities of the sources S_1 to S_9 be 5, 4, 2, 3, 0, 4, 1, 3 and 2 respectively, where 0 is the lowest and 5 is the highest priority. The sink and source node parameters are summarized in the Table 5.3.

The results are summarized in Table 5.4. From the table, we observe that none of the buffers in the system are fully consumed at any of the iterations, since the load considered for processing during the iterations $1, 3$ and 4 are insufficient to completely fill up the buffer of any of the sinks. Also, iteration 2 is forced to end early because of buffer space variations at sinks nodes. The values for $\alpha_{i,j}$ in Table 5.4 shall be used by the sinks to request loads from the sources. The values of $\alpha_{i,j}$ at iterations 1 to 4 are computed at time instants $t = 0,\ 3.689 \times 10^3,\ 10 \times 10^3$ and 11.786×10^3 seconds, respectively. The total processing time is $t = 15.24 \times 10^3$ seconds. The processing time and deadline requirements of the source nodes are as shown in Fig. 5.5 and are given in $\times 10^3$ seconds.

From Fig. 5.4 and 5.5, we make some interesting observations as follows.

Iteration 1: The load from the highest priority source S_1 is admitted into the system as its deadline could be satisfied. Having admitted S_1, the deadline requirements of S_1 and S_2 together could not be satisfied, and hence, the lower priority source S_2 is not admitted into the system. Then, S_3 is admitted into the system since the deadlines of S_1 and S_3 could be satisfied, provided S_1 is exclusively scheduled in this iteration. S_1 alone is scheduled in this iteration, since it has an earlier deadline requirement than S_3. At the end of this iteration, the processing for S_1 is completed.

Iteration 2: Though the source S_4 arrived at the system earlier, it is considered for processing only at the end of iteration 1. Since, its deadline could not be satisfied now, it is not admitted into the system. The source S_5 is admitted into

Table 5.3: Sink and Source node parameters for Example 5.2.

Sink nodes	**Parameter**	
	Inverse of computing speed (w_j)	**Buffer capacity (B_j)**
Sink node 1 (K_1)	1.11×10^{-9}	6 [at 0 sec]
		2 [at 10×10^3 sec]
		0 [at 20×10^3 sec]
Sink node 2 (K_2)	6.25×10^{-10}	2 [at 0 sec]
		3 [at 10×10^3 sec]
		1 [at 20×10^3 sec]
Sink node 3 (K_3)	5.00×10^{-10}	0 [at 0 sec]
		2 [at 10×10^3 sec]
		1 [at 20×10^3 sec]
Sink node 4 (K_4)	3.57×10^{-10}	3 [at 0 sec]
		1 [at 20×10^3 sec]

Source nodes	**Parameter**			
	Load Size (L_i)	**Load arrival time**	**Deadline**	**Priority**
Source node 1 (S_1)	3	0 sec	6×10^3 sec	5
Source node 2 (S_2)	4	0 sec	8×10^3 sec	4
Source node 3 (S_3)	3	0 sec	16×10^3 sec	2
Source node 4 (S_4)	2	2×10^3 sec	5×10^3 sec	3
Source node 5 (S_5)	5	3×10^3 sec	16×10^3 sec	0
Source node 6 (S_6)	2	9×10^3 sec	14×10^3 sec	4
Source node 7 (S_7)	2	8×10^3 sec	15×10^3 sec	1
Source node 8 (S_8)	5	11×10^3 sec	13×10^3 sec	3
Source node 9 (S_9)	1	11×10^3 sec	16×10^3 sec	2

Table 5.4: Load fraction and buffer utilization values for Example 5.2.

$\mathbf{q = 1}$	$\mathbf{S_1}$	$\mathbf{S_3}$		$\sum \alpha_{\mathbf{i,j}}^{\mathbf{(1)}}$	$\mathbf{B_j^{(1)}}$
K_1	0.5100	0.00		0.5100	5.4900
K_2	0.9050	0.00		0.9050	1.0950
K_4	1.5850	0.00		1.5850	1.4150
$\mathbf{q = 2}$	$\mathbf{S_3}$	$\mathbf{S_5}$		$\sum \alpha_{\mathbf{i,j}}^{\mathbf{(2)}}$	$\mathbf{B_j^{(2)}}$
K_1	0.3270	0.5450		0.8720	5.1280
K_2	0.5807	0.9678		1.5485	0.4515
K_4	1.0168	1.6947		2.7115	0.2885
$\mathbf{q = 3}$	$\mathbf{S_3}$	$\mathbf{S_5}$	$\mathbf{S_6}$	$\sum \alpha_{\mathbf{i,j}}^{\mathbf{(3)}}$	$\mathbf{B_j^{(3)}}$
K_1	0.00	0.00	0.2468	0.2468	1.7532
K_2	0.00	0.00	0.4382	0.4382	2.5618
K_3	0.00	0.00	0.5478	0.5478	1.4522
K_4	0.00	0.00	0.7672	0.7672	2.2328
$\mathbf{q = 4}$	$\mathbf{S_3}$	$\mathbf{S_5}$	$\mathbf{S_9}$	$\sum \alpha_{\mathbf{i,j}}^{\mathbf{(4)}}$	$\mathbf{B_j^{(4)}}$
K_1	0.1327	0.2212	0.1234	0.4773	1.5227
K_2	0.2356	0.3927	0.2191	0.8474	2.1526
K_3	0.2946	0.4910	0.2739	1.0595	0.9405
K_4	0.4126	0.6876	0.3836	1.4838	1.5162

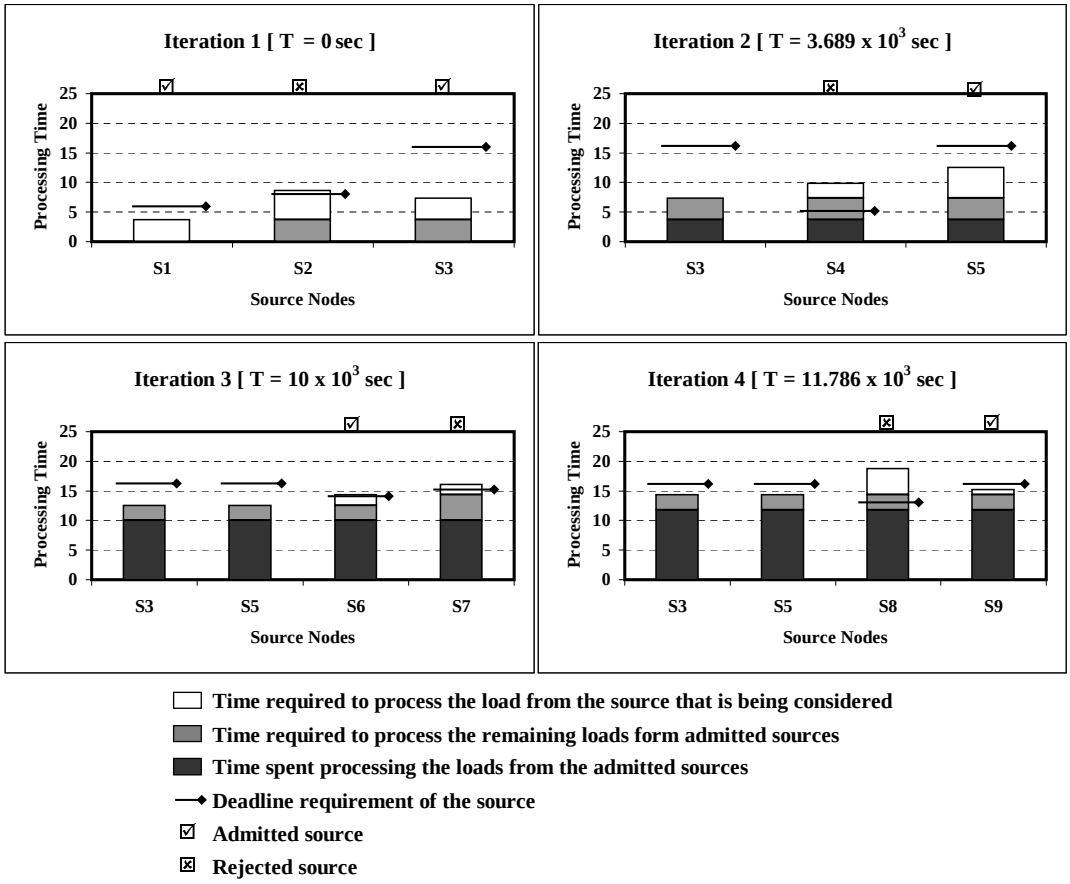

Figure 5.5: Performance of Progressive scheduling strategy in predictable time-varying buffer environment.

the system, since deadline of both S_3 (admitted in the previous iteration) and S_5 could be satisfied. Both these sources are scheduled for processing in this iteration, since their deadline requirements are the same.

Iteration 3: The source S_6 is admitted, since the system could satisfy the deadline requirements of the sources S_3, S_5 and S_6 together. The source S_7 is not admitted by the system, since having accepted S_3, S_5 and S_6 (which has higher priority than S_7), the deadline requirement of S_7 could not be satisfied by the system. In this iteration, processing of S_3 and S_5 are suspended and S_6 is scheduled exclusively since it has an earlier deadline than them. At the end of this iteration, the processing for S_6 is completed. It shall be noted that this does not violate the deadline requirements of S_3 and S_5.

Iteration 4: The source S_8 becomes inadmissible, since having accepted S_3 and S_5, the deadline requirement of S_8 could not be satisfied by the system. The source S_9 is admitted since the deadline requirements of S_3, S_5 and S_9 could be satisfied. All these sources are scheduled in this iteration, since they have the same deadline requirements. At the end of this iteration, the processing for S_3, S_5 and S_9 are all completed.

Thus, the Progressive scheduling scheme allows for temporarily suspending the processing of accepted loads, in order to completely process newly arrived loads with earlier deadlines. It considers the newly arrived load together with the loads that are already admitted into the system at the start of every iteration. This way of interleaving is one of the key characteristics of this approach as it ensures that

Table 5.5: Comparison of complexity of Resource aware distributed incremental scheduling (RADIS) strategies.

RADIS strategy	**Complexity**
Non-interleaved scheduling	$O(M+N)$
Earliest deadline first scheduling	$O(M+N^2)$
Progressive scheduling	$O(M+N^2)$

the loads admitted into the system are completely processed, irrespective of the priorities of the loads that arrive later. This technique also improves the system performance by admitting more sources. This can be realized by noting that in iteration 3, the source S_6 is admitted into the system and processed immediately. Here, S_3 and S_5, which are admitted into the system earlier, are temporarily suspended to allow resources to be utilized by S_6 to catch up its deadline. Another instance at which this can be realized is when the source S_3 is considered for admission in iteration 1. In this case, the source S_3 is admitted into the system only due to the fact that interleaving technique allows to schedule the source S_1 exclusively first.

5.2 Complexity of RADIS Strategies

The complexity for checking the feasibility for admitting new sources is $O(N^2)$ for EDF and Progressive scheduling strategies, and O(N) for Non-interleaved

Initial state:

$I = \{1, 2, ...N\}$, $J = \{1, 2, ...M\}$, $q = 0$, $T^{(0)} = 0$

Step 1: Determine the sources to be scheduled:

If ($X_{\text{new}} \neq \emptyset$) { **Check Feasibility for New Sources** }

If (($X_{\text{now}} = \emptyset$) & ($X_{\text{later}} \neq \emptyset$)) {

Step 1a: Non-interleaved Scheduling Scheme:

$X_{\text{now}} = X_{\text{later}}$, $X_{\text{later}} = \emptyset$

Step 1b: Earliest Deadline First Scheduling Scheme:

X_{now} = Source(s) with Earliest Deadline in $\{X_{\text{later}}\}$, $X_{\text{later}} = X_{\text{later}} - X_{\text{now}}$

Step 1c: Progressive Scheduling Scheme:

$Z_{\text{now}} = X_{\text{now}} \cup X_{\text{later}}$, $Z'_{\text{later}} = \emptyset$

Request & Receive the buffer availability estimation $\hat{B}_j^t$ from all the Sink nodes for the time $\max\{(T_{\text{d}_i} - T), S_i \in Z_{\text{now}}\}$.

$T_{\text{ul}} = T_{\text{cp}}$ / ({ $\sum_{j=1}^{M} \frac{1}{w_j}$, $\forall K_j$, where $\hat{B}_j^t \neq 0$, $j \in J$})

Step 1c(i): $Z_{\text{later}} = \emptyset$

$\hat{T} = T_{\text{ul}} \cdot \sum_{i=1}^{N} L_i$, $\forall S_i \in Z_{\text{now}}$, $i \in I$

{ If ($T_{\text{d}_i} > \hat{T}$) {$Z_{\text{later}} = Z_{\text{later}} \cup S_i$, $Z_{\text{now}} = Z_{\text{now}} - \{S_i\}$ } } , $\forall S_i \in Z_{\text{now}}$, $i \in I$

If ($Z_{\text{now}} = \emptyset$) { $X_{\text{now}} = Z_{\text{later}}$, $X_{\text{later}} = Z'_{\text{later}}$}

Else If ($Z_{\text{later}} = \emptyset$) {$X_{\text{now}} = Z_{\text{now}}$, $X_{\text{later}} = Z'_{\text{later}}$ }

Else { $Z'_{\text{later}} = Z'_{\text{later}} \cup Z_{\text{later}}$, Go to **Step 1c(i)**. } }

Step 2: Determine the buffer availability at Sink nodes:

If ($X_{\text{now}} \neq \emptyset$) { $P_{\text{now}} = P_{\text{all}}$

Receive ($\hat{B}_j^{(q+1)}$) from all Sink nodes.

If ($\hat{B}_j^{(q+1)} = 0$) $P_{\text{now}} = P_{\text{now}} - K_j$, $\forall K_j$, $j \in J$

Step 3: Determine $\alpha_j^{(q+1)}$ & $T^{(q+1)}$:

$\alpha_j^{(q+1)} = 1/(w_j \sum_{x=1}^{M} \frac{1}{w_x})$, $\forall K_j \in P_{\text{now}}$, $j \in J$

$L = \sum_{i=1}^{N} L_i$, $\forall S_i \in X_{\text{now}}$, $i \in I$

$Y = \min\{\hat{B}_j^{(q+1)}/(\alpha_j^{(q+1)} L)$, $\forall K_j \in P_{\text{now}}$, $j \in J\}$

If ($Y > 1$) {$Y = 1$}

$T^{(q+1)} = Y \alpha_j^{(q+1)} L\ w_j\ T_{\text{cp}}$, *for any* K_j , $j \in J$

Broadcast the schedule information (Y, $\alpha_j^{(q+1)}$, L, $(T + T^{(q+1)})$, (L_i , $\forall S_i \in X_{\text{now}}$, $i \in I$)) to all the Sink nodes.

Step 4: Update the amount of load remaining to be processed:

Wait till $(T + T^{(q)})$ & Receive (L_j) from all the Sink nodes.

$q = q + 1$

$L_i = L_i \cdot (1 - Y + ((\sum_{j=1}^{M} L_j)/L))$, $\forall S_i \in X_{\text{now}}$, $i \in I$

{ If ($L_i = 0$) {$X_{\text{now}} = X_{\text{now}} - \{S_i\}$}}, $\forall S_i \in X_{\text{now}}$, $i \in I$ }

Go to **Step 1**.

Figure 5.6: Pseudo code describing the workings of the RADIS scheduler at the coordinator node C_s.

Check Feasibility for New Sources
{

Step 1: Initialization:

S_i = Highest priority source in $\{X_{\text{new}}\}$.

$X_{\text{new}} = X_{\text{new}} - \{S_i\}$, $Z_{\text{now}} = X_{\text{now}} \cup X_{\text{later}} \cup S_i$

Request & Receive the buffer availability estimation $\hat{B}_j^t$ from all the Sink nodes for the time $(\max\{T_{\text{d}_i} - T\}, S_i \in Z_{\text{now}})$.

$T_{\text{ul}} = T_{\text{cp}}$ / ({ $\sum_{j=1}^{M} \frac{1}{w_j}$, $\forall K_j$, where $\hat{B}_j^t \neq 0$, $j \in J$})

Step 2: Non-interleaved Scheduling Scheme:

$\hat{T} = T_{ul} \cdot \sum_{i=1}^{N} L_i$, $\forall S_i \in Z_{\text{now}}$, $i \in I$

{ If $(T_{\text{d}_i} > \hat{T})$ { $Z_{\text{now}} = Z_{\text{now}} - \{S_i\}$ } } , $\forall S_i \in Z_{\text{now}}$, $i \in I$

Step 2a: Admit the new source:

If $(Z_{\text{now}} = \emptyset)$ { $X_{\text{later}} = X_{\text{now}} \cup S_i$, $X_{\text{now}} = \emptyset$}

Step 2b: Reject the new source:

Else { Message **"No feasible solution for S_i."** }

Step 3: Earliest Deadline First Scheduling Scheme:

While $(Z_{\text{now}} \neq \emptyset)$ {

S_i = Earliest Deadline Source in $\{Z_{\text{now}}\}$.

$\hat{T} = \hat{T} + T_{ul} \cdot L_i$

If $(T_{\text{d}_i} > \hat{T})$ { $Z_{\text{now}} = Z_{\text{now}} - \{S_i\}$ }

Else { break; } }

Step 3a: Admit the new source:

If $(Z_{\text{now}} = \emptyset)$ { $X_{\text{later}} = X_{\text{now}} \cup X_{\text{later}} \cup S_i$, $X_{\text{now}} = \emptyset$}

Step 3b: Reject the new source:

Else { Message **"No feasible solution for S_i."** }

Step 4: Progressive Scheduling Scheme:

$\hat{T} = T_{\text{ul}} \cdot \sum_{i=1}^{N} L_i$, $\forall S_i \in Z_{\text{now}}$, $i \in I$

$Z_{\text{later}} = \emptyset$

{ If $(T_{\text{d}_i} > \hat{T})$ { $Z_{\text{later}} = Z_{\text{later}} \cup S_i$, $Z_{\text{now}} = Z_{\text{now}} - \{S_i\}$ } } , $\forall S_i \in Z_{\text{now}}$, $i \in I$

Step 4a: Admit the new source:

If $(Z_{\text{now}} = \emptyset)$ { $X_{\text{later}} = X_{\text{now}} \cup X_{\text{later}} \cup S_i$, $X_{\text{now}} = \emptyset$}

Step 4b: Reject the new source:

Else If $(Z_{\text{later}} = \emptyset)$ { Message **"No feasible solution for S_i."** }

Step 4c: Consider the sources in Z_{now} and re-iterate:

Else Go to **Step 4**.

}

Figure 5.7: Pseudo code describing the workings of the admission control procedure at the coordinator node C_s.

Initial state:

$q = 0$, $T^{(0)} = 0$, $p = 0.95$

Step 1: Buffer availability estimation for next iteration:

$\hat{B}_j^{(q+1)} = ((\sum_{k=0}^{(s-1)} ((s-k) \cdot B_j^{(q-k)})) / (\sum_{k=0}^{(s-1)} k)) * p$

Send $(\hat{B}_j^{(q+1)})$ to the Coordinator node.

Step 2: Wait till the previous iteration is completed:

Wait till $(T + T^{(q)})$.

Step 3a: Compute Load amounts to be processed:

Receive the schedule information (Y, $\alpha_j^{(q+1)}$, L, $(T + T^{(q+1)})$, $(L_i, \forall S_i \in X_{\text{now}}, i \in I)$) from the Coordinator node.

$q = q + 1$

If $(\hat{B}_j^{(q)} \neq 0)$ {

  If $((Y\alpha_j^{(q)} L) > B_j^{(q)})$ { $B_j^{(q)} = 0$

    $L_j = (Y\alpha_j^{(q)} L) - B_j^{(q)}$

    $\alpha_{i,j}^{(q)} = L_i \cdot \frac{B_j^{(q)}}{L}$, $\forall S_i \in X_{\text{now}}$, $i \in I$ }

  Else { $L_j = 0$

    $B_j^{(q)} = B_j^{(q)} - (Y\alpha_j^{(q)} L)$

    $\alpha_{i,j}^{(q)} = Y\alpha_j^{(q)} L_i$, $\forall S_i \in X_{\text{now}}$, $i \in I$ }

  Send (L_j) to the Coordinator node.

Step 3a(i): Schedule the loads from Source Nodes:

  Request, receive and process the load fractions $(\alpha_{i,j}^{(q)})$ from the Source Nodes $S_i \in X_{\text{now}}$.

}

Go to **Step 1**.

Step 3b: Buffer availability estimation for the time t_{req}:

Receive the buffer availability estimation request from the Coordinator node for the time t_{req}.

$\hat{B}_j^t = \frac{1}{t_{\text{req}}} \int_{t=(T-t_{\text{req}})}^{t=T} B_j(t)\, dt$, where $(T - t_{\text{req}}) \geq 0$

Send $(\hat{B}_j^t)$ to the Coordinator node.

Go to **Step 2**.

Figure 5.8: Pseudo code describing the workings of the RADIS scheduler at the sink nodes.

scheduling strategy. The complexity for determining the sources to be scheduled in an iteration is $O(N^2)$ for EDF and Progressive scheduling strategies and O(1) for Non-interleaving scheduling strategy. The complexity for the calculation of scheduling information per iteration is $O(M)$ for all the scheduling strategies. Hence, the complexity of RADIS with Non-interleaving scheduling strategy is $O(M+N)$, and the complexity of RADIS with EDF and Progressive scheduling strategies is $O(M+N^2)$. The complexity comparisons are summarized in the Table 5.5.

5.3 Performance Evaluation

Now, we shall describe our experimental platform and list certain parameters that influence the performance of the methodologies. In our study, we simulated 64, 128 and 256 node (sink) systems. The sink speed ($\frac{1}{w_j}$) parameters are derived from the STAR experiments conducted at BNL [51]. The speed parameter is assigned to the sink nodes based on uniform probability distribution.

We allow the buffer availabilities at the sinks to vary randomly over time (referred to as *Time-varying Buffer* (TVB) scenario), and, we also observe the best case performance when the buffer sizes are time-invariant (referred to as *Time-invariant Buffer* (TIB) scenario). The TIB scenario results are important as they provide the upper bounds for the performance of the strategies. Also, all possible variations (for priority of loads and available buffer sizes) ranging from small to large fluctu-

ations and the frequency of buffer availability fluctuations are captured. These are varied randomly following uniform probability distributions while the load arrival rates follow Poisson distribution. In our simulations, we consider three different sets of loads. Set 1 consists of 10% Type I and 90% Type II load sizes (see Table 5.6). Set 2 consists of 50% Type I and 50% Type II load sizes. Set 3 consists of 90% Type I and 10% Type II load sizes. All the above mentioned simulation parameters and their respective ranges, are given in Table 5.6.

In our experiments, the number of sinks participating in every iteration is also allowed to vary simulating the nodes leaving the system / participating in the computation in a random fashion. Thus we attempt to capture the behavior of the strategies close to a real-life environment. Further, our schemes guarantee that the deadlines for all the loads that are accepted will be satisfied, since in our experiments the buffer sizes follow uniform probability distribution and our schemes utilize time averaged buffer estimation strategy for admissibility testing.

5.3.1 Metrics of Interest

We consider the following performance metrics that are of direct relevance to this study. The number of loads accepted in the TVB scenario (Φ) has higher significance, since one of our aim is to maximize the number of loads that are accepted. We also define the acceptance ratio (β), and the ratio of acceptance

Table 5.6: Simulation parameters and their range of values.

Parameter	**Range of values**
Simulation period [seconds]	$50,000$
Load arrival rates (λ) [arrivals/second]	$0.001 - 1.0$
Number of Sinks (M)	$64,\ 128,$ and 256
Window size for Buffer Estimation Strategy (s)	8
Inverse of Sink Speeds (w_j)	1.11×10^{-9}, 6.25×10^{-10}, 5.00×10^{-10} and 3.57×10^{-10}
Computing Intensity Constant (T_{cp})	6.52×10^{10}
Load Sizes (Type I \ Type II)	$(50 - 70)\backslash(170 - 190)$
Load Deadlines (Type I \ Type II) [seconds]	$(500 - 750)\backslash(6000 - 7000)$
Priority of Sources	$0 - 10$
Buffer Sizes (Small\Medium\Large)	$(0,\ 0.5)\backslash(0,\ 0.75)\backslash(0,\ 0.5,\ 1.0)$
Buffer Size Variations (Slow\Medium\Fast)	$(3000-3500)\backslash(2000-2500)\backslash(1000-1500)$

ratios (η) as,

$$\begin{aligned} \beta &= \frac{Number\ of\ loads\ accepted}{Number\ of\ loads\ arrived} \\ \eta &= \frac{\beta_{\text{TVB}}}{\beta_{\text{TIB}}} \end{aligned} \tag{5.9}$$

where β_{TVB} and β_{TIB} are the acceptance ratios of TVB and TIB scenarios respectively.

Secondly, for the TVB scenario, we define a metric (γ) that quantifies the throughput of the system as,

$$\gamma = \frac{Number\ of\ loads\ processed}{Number\ of\ loads\ accepted} \tag{5.10}$$

Finally, we define the average buffer utilization in an iteration at a sink node (ζ), and the ratio of the average buffer utilization (χ) as,

$$\begin{aligned} \zeta &= \frac{\sum_1^M \left(\left(\sum_{i=1}^q \frac{\min\{Y\alpha_j^{(i)}L,\ B_j^{(i)}\}}{B_j^{(i)}} \right) \ / \ q \right)}{M} \\ \chi &= \frac{\zeta_{\text{TVB}}}{\zeta_{\text{TIB}}} \end{aligned} \tag{5.11}$$

where q is the number of iterations that the sink node has participated and ζ_{TVB} and ζ_{TIB} are the average buffer utilization in an iteration at a sink node in TVB and TIB scenarios respectively.

5.3.2 Discussion of the Results

The behavior of Φ , β_{TVB} , η , γ and χ when we employ RADIS in a system with 64, 128 and 256 sinks with respect to the load arrival rates for two different deadline types as given in the Table 5.6 are plotted in Fig. 5.9, 5.10, and 5.11 respectively. In these plots, we denote the Non-interleaved Scheduling Scheme with Load Sets 1, 2 and 3 as $NS1$, $NS2$ and $NS3$, the Earliest Deadline First Scheme with Load Sets 1, 2 and 3 as $ES1$, $ES2$ and $ES3$, and the Progressive Scheme with Load Sets 1, 2 and 3 as $PS1$, $PS2$ and $PS3$ respectively. Here, we analyze based on the simulation results for the 64 node system shown in Fig. 5.9, since all our strategies exhibit similar trend for all the performance metrics considered irrespective of the number of sinks in the system.

From the Fig. 5.9, it is observed that at low arrival rates (less than 0.006) there is little difference in Φ for the various scheduling schemes. As the arrival rate increases, irrespective of the scheduling schemes and the Deadline Type of the loads, the number of loads accepted for Load Set 3 is higher than that for Load Set 2, which in turn is higher than that for Load Set 1. It is also interesting to observe that a better performance is shown by the Progressive Scheduling Scheme in the case of loads with Type I deadlines where as by the EDF Scheduling Scheme for the loads with Type II deadlines. Also, the performance improvement at higher arrival rates for loads with Type II deadlines is significantly higher for the EDF scheme when compared with other schemes. But, the improvement is of less significance for arrival rates higher than 0.3, since the β_{TVB} is closer to 0 at these rates (See

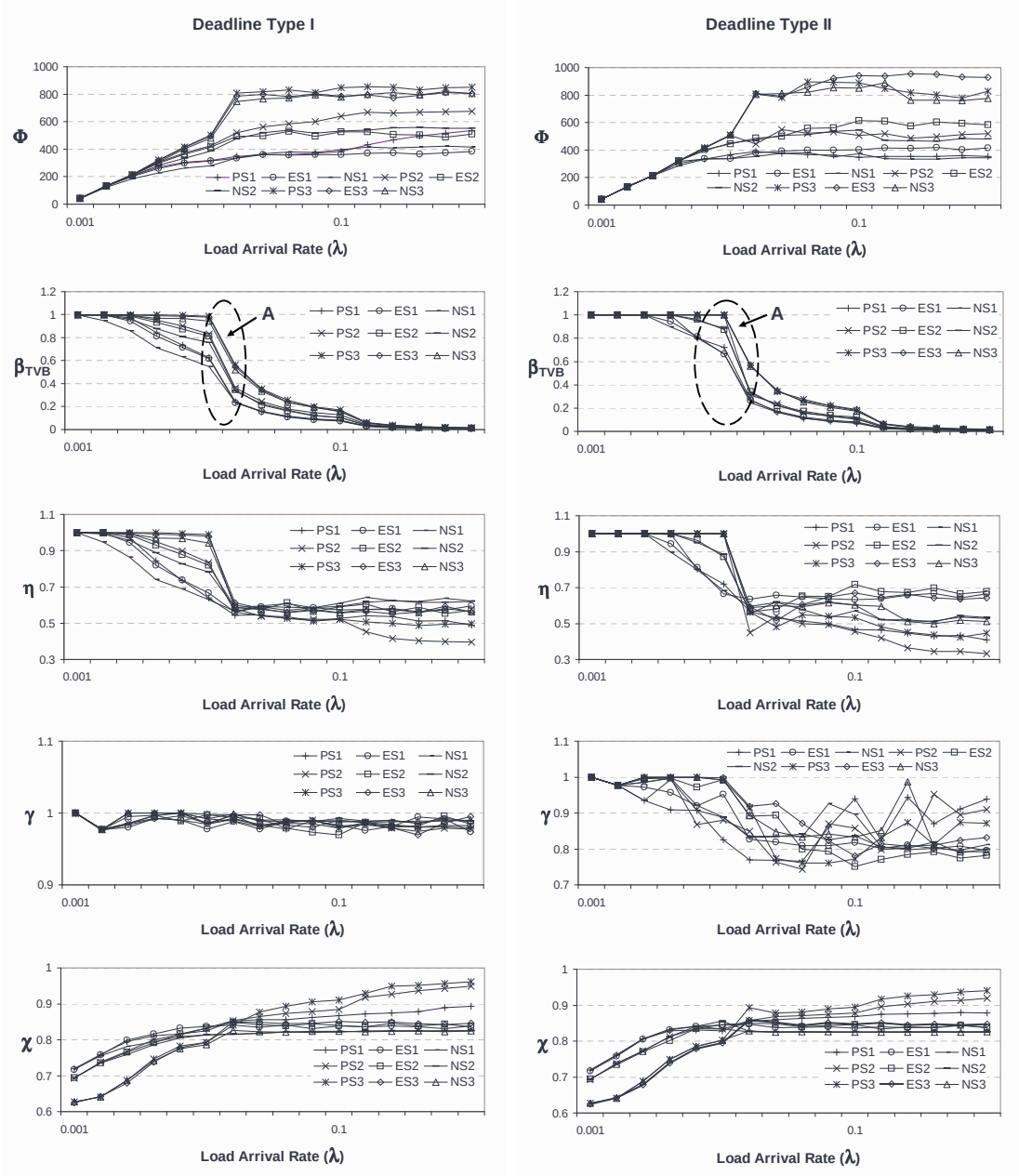

Figure 5.9: Simulation results for the number of loads accepted (Φ) and the load acceptance ratio (β_{TVB}) in the time-varying buffer (TVB) scenario; the ratio of acceptance ratios (η) and the ratio of the average buffer utilization (χ) in TVB and time-invariant buffer (TIB) scenarios; and throughput of the system (γ) for the non-interleaved, earliest deadline first, and progressive interleaved scheduling RADIS schemes for load sets 1, 2, and 3 and deadline types I and II in a 64-node cluster system.

plot of β_{TVB} in Fig. 5.9).

Initially, all the arriving loads get accepted by the system (the acceptance ratio is close to 1) and as the arrival rate increases further, it starts falling steeply (the zone represented as 'A' in the plot of β_{TVB} in Fig. 5.9). This is due to the fact that the scheduler can no longer continue to accept the newly arrived loads unless the deadline requirements of already accepted loads together with the new load being considered could be satisfied. Hence, the admissibility testing starts rejecting some of the newly arrived loads. As the arrival rate further increases, the acceptance ratio β_{TVB} moves closer to 0.

From the plot of η in Fig. 5.9, it is observed that at arrival rates lower than 0.03 for all Load Sets and Deadline Types, η is almost similar for both the EDF and Progressive schemes of RADIS, because the acceptance ratios (β) of these schemes are almost identical. At these arrival rates for loads with Type I deadlines, the η values for both the EDF and Progressive schemes of RADIS are higher than that for the Non-interleaved Scheduling Scheme. At higher arrival rates, for all load sets with Type I deadlines, η tends to saturate close to a value of 0.6 for the EDF and Non-interleaved schemes and about $0.4 - 0.5$ for the Progressive Scheme, where as, for all load sets with Type II deadlines, η tends to saturate close to a value of 0.65 for the EDF Scheme, 0.5 for the Non-interleaved Scheme and around $0.3 - 0.4$ for the Progressive Scheme.

It is also observed that the system throughput γ value is closer to 1 for loads with Type I deadline requirements irrespective of the load arrival rates, where as it

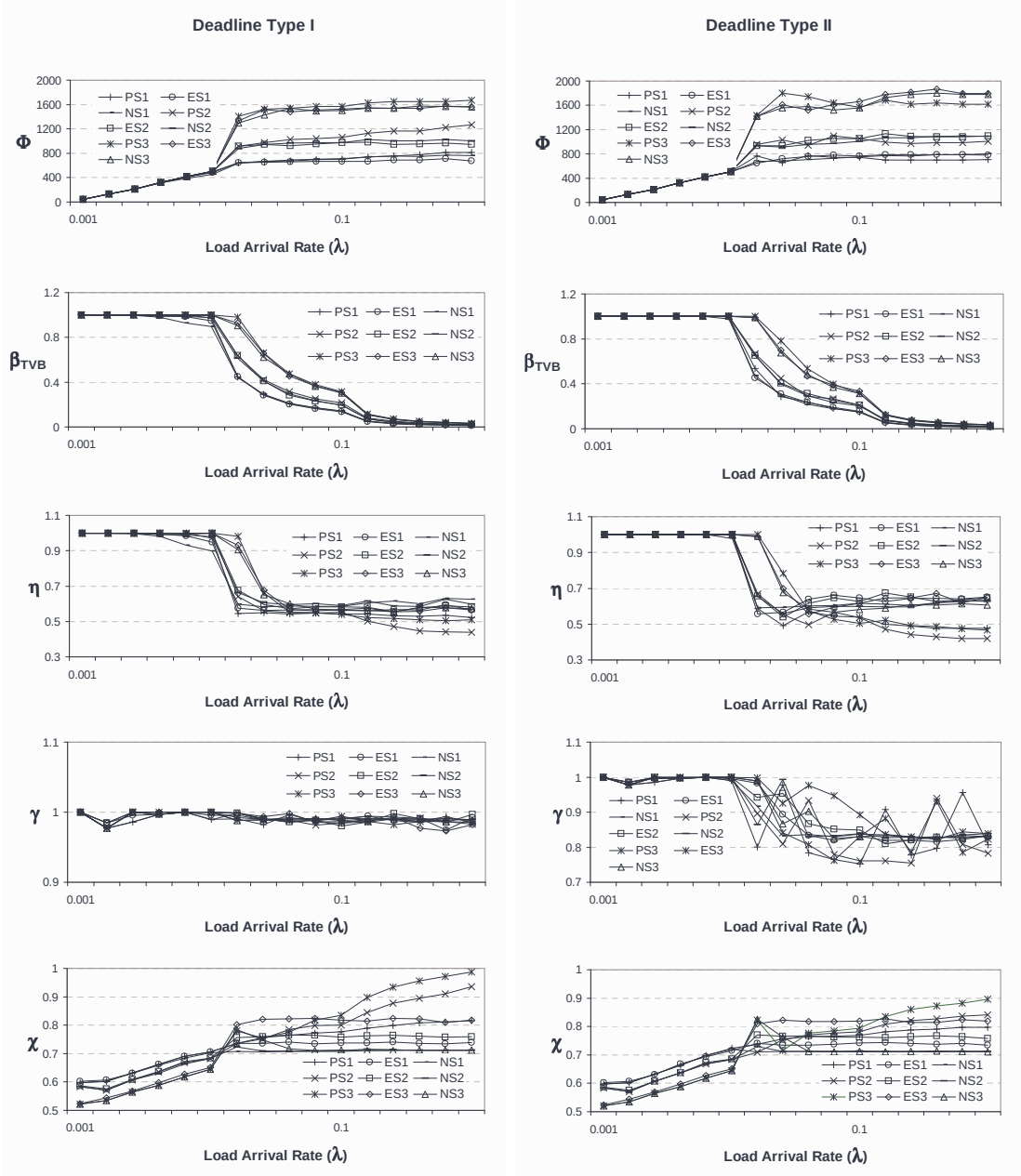

Figure 5.10: Simulation results for the number of loads accepted (Φ) and the load acceptance ratio (β_{TVB}) in the time-varying buffer (TVB) scenario; the ratio of acceptance ratios (η) and the ratio of the average buffer utilization (χ) in TVB and time-invariant buffer (TIB) scenarios; and throughput of the system (γ) for the non-interleaved, earliest deadline first, and progressive interleaved scheduling RADIS schemes for load sets 1, 2, and 3 and deadline types I and II in a 128-node cluster system.

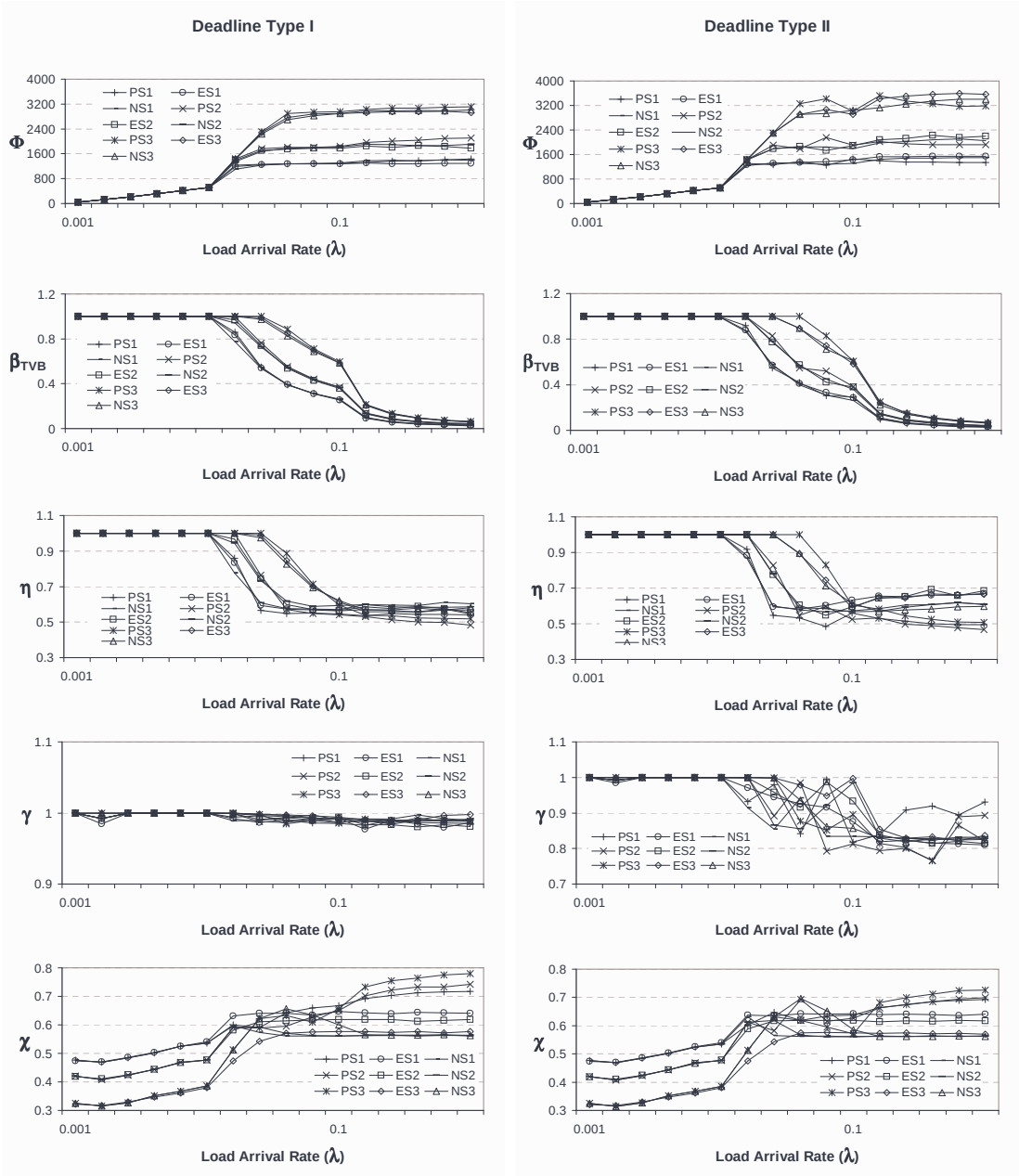

Figure 5.11: Simulation results for the number of loads accepted (Φ) and the load acceptance ratio (β_{TVB}) in the time-varying buffer (TVB) scenario; the ratio of acceptance ratios (η) and the ratio of the average buffer utilization (χ) in TVB and time-invariant buffer (TIB) scenarios; and throughput of the system (γ) for the non-interleaved, earliest deadline first, and progressive interleaved scheduling RADIS schemes for load sets 1, 2, and 3 and deadline types I and II in a 256-node cluster system.

decreases with increasing arrival rates for loads with Type II deadline requirements for all the load sets and schemes. In the case of the EDF Scheme, though the system throughput decreases with the increase in load arrival rates for loads with Type II deadlines, it is seen to be more robust, since the variations in the system throughput are lesser compared with other schemes (Refer to the plot of γ in Fig. 5.9).

The plot of χ in Fig. 5.9 shows that at arrival rates lower than 0.03 for both Type I and Type II deadline requirements of the loads, the average buffer utilization for all the schemes are almost identical and the utilization of Load Set 1 is higher than that of Load Set 2 and the utilization of Load Set 2 is higher than that of Load Set 3. For arrival rates higher than 0.03, for the both the deadline types and all the load sets the utilization saturates at a value of around 0.8 in the case of both the EDF and Non-interleaved Scheduling schemes, where as, the trend reverses in the case of the Progressive Scheme and then on the utilization of Load Set 3 is higher than that of Load Set 2 and the utilization of Load Set 2 is higher than that of Load Set 1 and the values saturate between $0.85 - 0.95$ at higher arrival rates.

It is to be noted that at arrival rates lower than 0.006, the number of loads accepted for all the schemes are almost the same and at higher arrival rates although the acceptance ratios of all the schemes are almost similar, for loads with Type I deadlines, the average buffer utilization is higher and the number of loads that are accepted are also higher in the case of the Progressive scheme, where as for loads with Type II deadlines, the average buffer utilization is lower and the number of

loads that are accepted are higher in the case of the EDF scheme.

In an actual system we propose to have a decision making mechanism that monitors the load arrival patterns and their deadline requirement at the coordinator node C_s and dynamically choose the appropriate scheduling scheme. Hence, irrespective of type of loads and their deadline requirements, when the load arrival rate is lower, we propose to utilize the simpler Non-interleaved Scheduling Scheme. When the load arrival rate increases further, depending up on the type of deadline requirements of the loads, we propose to utilize either the Progressive or the EDF Scheduling schemes. However, when the load arrival rate increases further and the acceptance ratio (β_{TVB}) is closer to 0 (when $\lambda > 0.3$, in the 64 node cluster system), we propose to utilize the simpler Non-interleaved Scheduling Scheme. Thus, all the proposed schemes are very useful for real-life systems.

Chapter 6

Strategies for Scheduling across Cluster Systems

Computational Grid systems comprise of interconnected clusters. In the previous chapters, strategies for scheduling with in clusters are presented. In this chapter, we shall propose and analyze load distribution strategies for scheduling across clusters that are interconnected to form a backbone network.

The backbone network in a computational Grid system comprises of the master nodes of the clusters forming an arbitrary topology/graph $G = \langle C, E \rangle$, where C denotes the number of master nodes interconnected via E communication links. We assume a uni-port communication model and that all the nodes in this backbone system have front-ends. We consider scheduling when the loads with deadline requirements arrive at arbitrary times to a dynamic system wherein the nodes are

allowed to join or leave the system. Our strategies assume that the nodes participating in computation for an accepted load, shall not leave the system until the processing is completed for the load portions assigned to it as well as the sub-tree for which it is a parent, else the guarantees given to the loads while admitting may not be fulfilled. We also assume that all the processing nodes shall allow their buffers to be reclaimed for multi-installment distribution until the processing for the accepted load portions are completed.

In a real life system, there shall be multiple source nodes in the system at a given time and also the loads may arrive at them dynamically. Hence, we shall consider the source nodes based on their priority order and choose one of them as the root node. At this chosen root node, our proposed distribution strategies shall be executed to determine the load fractions to be distributed to the processing nodes. If there are multiple load arrivals at the root node, it shall also consider them based on their (load's) priority order.

England et al [59] have proven that the optimal solution to single-installment based divisible load scheduling problem on a arbitrary graph indeed occurs on a spanning tree of the graph, a multi-level tree. Though our strategies distribute load in multiple installments for a given load, we first compute the optimal fractions based on single installment and then distribute the load in multi-installments based on the buffer availability at the processing nodes, and also assume that the system parameters (buffer availability at nodes, number of participating nodes etc) do not change until the processing is completed for the accepted load. Hence, the solution

for our problem shall also occur on a spanning tree.

For an arbitrary graph network G of the backbone network of a Grid system, we shall first generate a spanning tree. For an arbitrary graph, there normally exist many spanning trees. Also, Byrnes et al [60] have proved that finding the optimal spanning tree (the spanning tree that generates minimum total processing time) on the arbitrary network is NP-hard. Therefore, one immediate question to address is which spanning tree(s) deliver efficient solution for the load distribution strategies described in Chapter 2. We present the spanning tree construction strategies in the Section 6.1, and our Resource Aware Sequential Load Distribution (RASLD) and Resource Aware Parallel Load Distribution (RAPLD) strategies are described in Section 6.2 and 6.3 respectively.

6.1 Spanning Tree Construction Strategies

In this section, we present the various spanning tree construction strategies that we shall apply with our distribution strategies and their characteristics in brief.

Minimum spanning tree (MST): In MST, the total link weight (the link weights depend on the link delays) is the minimum among all the spanning trees. Since MST always tends to incorporate the link with small weight without considering its hop count to the root, normally MSTs are very deep and "skinny". Kruskal's or Prim's algorithm [62] are used to construct such a spanning tree.

Shortest path spanning tree (SPT): In SPT, each node has the shortest path (in terms of link weights) to the root. To construct such a spanning tree, either the efficient Dijkstra's or Bellman-Ford's algorithm [62] could be used. The shape of the tree depends on the distribution of the link weights. The SPT trees are generally deeper and have smaller node degrees than FHT trees.

Fewest hops spanning tree (FHT): In FHT, each node's hop count to the root is the minimum. The breadth-first search (BFS) algorithm [63] could be used to construct the FHT. FHTs tend to be shallow and "fat".

Robust spanning tree (RST): RST [59] is designed to seek a trade-off between link weight and hop count. Such a tree is immune to data loss when nodes or links fail and yet provides good performance. RST minimizes each node's combined cost of link weight and hop count as follows.

$$\lambda * hop\ count + (1 - \lambda) * link\ weight \tag{6.1}$$

The weight λ is actually a function of a node's depth in the tree, which falls into the range $[0, 1)$ When an edge (i, j) is being considered for inclusion in the tree, then

$$\lambda_i = 1 - \frac{h_i}{\epsilon_1} \tag{6.2}$$

where i is the new vertex not already in the tree, h_i is the hop count of node i from the root and ϵ_1 is the depth of the deepest leaf in the shortest path spanning tree (SPT) or in other words it is the deepest of the shortest paths from the root

node to all other nodes in the network, and this gives the relative importance of hop count versus link weights. RST strives for a balance between SPT and FHT.

All these spanning tree construction strategies consider either the link delays (link weights) or hop counts where as our load distribution strategies in Chapter 3 consider both processing speed of the nodes as well as link delays. Hence, here in we propose a spanning tree construction strategy that considers both these parameters and thereby strives to provide a minimum optimal processing time for the given network.

Minimum network equivalence spanning tree (EST): Our EST strategy assumes optimal sequencing load distribution and maximizes the equivalent computation power of the spanning tree by considering both processor and the link weights (or speeds) while constructing the spanning tree. The EST spanning tree construction algorithm uses the equivalent processor model described in Chapter 3. Given an arbitrary network (G) containing nodes (N) and links (E), it first adds the root node to the spanning tree and then, considers all the links originating from this spanning tree, one by one, and adds the (E, N) pair that provides minimum effective equivalent processor value ($w_{eq(0)}$) (as detailed in the Chapter 3) to it and continues until all the nodes in G are added. The shape of this spanning tree depends on the distribution of the link speed as well as processing speed of the nodes.

6.2 Resource Aware Sequential Load Distribution Strategy

In this section, we shall present our RASLD strategy, that follows the load distribution strategy described in Chapter 3. For sequential distribution, sum of link delays (communication delays) along the path from the root node to processing nodes is computed, and a single-level tree is derived with the computed sum as the link delay value for the link between the root node and that processing node. In SPT, each node has the shortest path (in terms of link weights) to the root. Hence, it can be deduced that SPT will provide the best solution for sequential load distribution. Hence, our RASLD strategy utilizes SPT algorithm and computes the load distribution following the optimal sequencing [16] as explained in Fig. 6.1.

It shall be noted that RASLD strategy adapts to the real life scenario where in the nodes in the system (including the load originating or root node) may or may not participate in processing the load but could communicate or relay the load portions to their child nodes for processing. The RASLD strategy also adapts to scenarios where in there are time critical loads and also finite buffer capacity constraints at the processing nodes. When there are time critical loads, admissibility testing shall be performed based on the deadline requirements of the load and the computed optimal processing time by the RASLD algorithm and decide on whether to accept or reject it. If the buffer available at a processing node

Step 1: Select the nodes for load distribution:

Given an arbitrary network G and a root node, say C_0, drop the isolated nodes (nodes that could not be communicated with).

Step 2: Spanning tree construction:

Construct the Shortest path spanning tree G_{SPT} from the root node to the selected nodes in the arbitrary network G.

Step 3: Reduce the multi-level tree to a single-level tree:

For all the nodes in G_{SPT}, determine the sum of link delays from the root node to each of them and "Replace" G_{SPT} with a single-level tree with root node as C_0 and all the other nodes, that could process the load, connected to it with links having delays equivalent to the sum of the link delays thus determined for them. Note that this ensures that the load will not be assigned to the child nodes that could only communicate.

Step 4: Reduce the single-level tree to a single equivalent node:

For the single-level tree network $\Sigma(0,(m+1))$, apply Theorem 1 and identify the optimal sequence for load distribution.

If C_0 could process the load as well as communicate with other nodes, determine the equivalent speed parameter $w_{eq(0)}$ using (3.11) for the equivalent node $C_{eq(0)}$ and "Replace" the single-level tree with the equivalent node $C_{eq(0)}$.

Else if C_0 could only communicate with other nodes, determine the equivalent speed parameter $w_{eq(0)}$ using (3.17) for the equivalent node $C_{eq(0)}$ and "Replace" the single-level tree with the equivalent node $C_{eq(0)}$.

Step 5: Load distribution with optimal sequence:

Inflate $C_{eq(0)}$ to obtain the single-level tree $\Sigma(0,m+1)$ and determine $\alpha^*_{0,i}$, using (3.12) if C_0 could process the load, or (3.18) if C_0 could only communicate with other nodes, and distribute the load to all the nodes in this tree following the optimal sequence.

Optimal Load Fractions: $L^*_{x,i} = \alpha^*_{0,i} \cdot L$;

Optimal Processing Time: $T^*(\alpha^*) = T(\Sigma(0,m+1)) = T^*_{x,i}(\alpha^*)\ ,\ \forall\ C_{x,i} \in G_{\mathrm{SPT}}$.

Figure 6.1: Resource aware sequential load distribution algorithm (RASLD).

is insufficient to accommodate the computed single installment load fractions for that node, the load shall be distributed in multiple installments. As in the case of load distribution within clusters described in Chapter 4 and 5, here again the load that shall be distributed in each installment to every processing node shall be computed based on the computed load fractions and the buffer availability among all the processing nodes as given by (3.6) and (5.1).

6.3 Resource Aware Parallel Load Distribution Strategy

In this section, we propose our RAPLD strategy that follows the load distribution strategy described in Chapter 3. Given an arbitrary graph network G with root node, say C_0, our strategy drops the isolated nodes in G thus selecting the nodes that shall participate in the computation and generates a spanning tree for it based on the chosen construction strategy (Section 6.1). Then the multi-level tree is systematically reduced to a single-level tree by replacing the sub-trees with their equivalent nodes and compute the load distribution following the optimal sequence theorem [16] as explained in Fig. 6.2.

The RAPLD strategy also adapts to the real life scenario where in the nodes in the system (including the load originating or root node) may or may not participate in processing the load but could communicate or relay the load portions to their child nodes for processing. Like RASLD strategy, it also adapts to scenarios where

Step 1: Select the nodes for load distribution:

Given an arbitrary network G and a root node, say C_0, drop the isolated nodes (nodes that could not be communicated with).

Step 2: Spanning tree construction:

Construct a spanning tree G_{Tree} from the root node to the selected nodes in the arbitrary network G. Drop the leaf nodes (nodes with no children) that could only communicate in G_{Tree}.

Step 3: Reduce the multi-level tree to a single-level tree:

for ($j = Q;\ j > 1;\ j--$) { // Q is the total number of levels in G_{Tree}

  for ($r = 1;\ r \leq R;\ r++$) { // R is the total number of sub-trees at level j

    In this single-level sub-tree with, say $C_{x,i}$, as parent node, apply Theorem 1 and identify the optimal sequence for load distribution and determine the equivalent speed parameter $w_{x,eq(i)}$ using (3.17) for the equivalent node $C_{x,eq(i)}$ of this sub-tree.

    If $C_{x,i}$ could process the load as well as communicate with other nodes, "Replace" the single-level sub-tree with two nodes, the node $C_{x,i}$ and the equivalent node $C_{x,eq(i)}$, both connected to the parent of $C_{x,i}$ with link delay values of $z_{x,i}$. The two nodes shall be arranged in such an order that, while distributing the loads to them, their parent node distributes the load to first $C_{x,i}$ and then to $C_{x,eq(i)}$.

    Else if $C_{x,i}$ could only communicate with other nodes, "Replace" the single-level sub-tree with only one node, the equivalent node $C_{x,eq(i)}$, connected to the parent of $C_{x,i}$ with a link delay value of $z_{x,i}$.

  }

}

Step 4: Reduce the single-level tree to a single equivalent node:

For the single-level tree network $\Sigma(0,(m+1))$, apply Theorem 1 and identify the optimal sequence for load distribution.

If C_0 could process the load as well as communicate with other nodes, determine the equivalent speed parameter $w_{eq(0)}$ using (3.11) for the equivalent node $C_{eq(0)}$ and "Replace" the single-level tree with the equivalent node $C_{eq(0)}$.

Else if C_0 could only communicate with other nodes, determine the equivalent speed parameter $w_{eq(0)}$ using (3.17) for the equivalent node $C_{eq(0)}$ and "Replace" the single-level tree with the equivalent node $C_{eq(0)}$.

Step 5: Load distribution with optimal sequence:

Inflate $C_{eq(0)}$ to obtain the single-level tree $\Sigma(0,m+1)$ and determine $\alpha^*_{0,i}$, using (3.12) if C_0 could process the load, or (3.18) if C_0 could only communicate with other nodes, and distribute the load to all the nodes in this tree following the optimal sequence.

For every node in a level j, $j = 1,2,...,Q$, determine $\alpha^*_{x,i}$, using (3.18) and assign it to them. Inflate every equivalent node $C_{eq(i)}$, in this level j, and distribute the load assigned to $C_{eq(i)}$, following an optimal sequence among the nodes that formed $C_{eq(i)}$ using (3.18).

Optimal Load Fractions: $L^*_{x,i} = \alpha^*_{x,i} \cdot L$;

Optimal Processing Time: $T^*(\alpha^*) = T(\Sigma(0,m+1)) = T^*_{x,i}(\alpha^*)\ ,\ \forall\ C_{x,i} \in G_{\text{Tree}}$.

Figure 6.2: Resource aware parallel load distribution algorithm (RAPLD).

in there are time critical loads and also finite buffer capacity constraints at the processing nodes. When there are time critical loads, admissibility testing shall be performed based on the deadline requirements of the load and the computed optimal processing time for it by the RAPLD algorithm and the decision be made to accept or reject it. If the buffer available at a processing node is insufficient to accommodate the computed single installment load fractions for that node, the load shall be distributed in multiple installments. As with all the strategies proposed in this thesis, here again the load that shall be distributed in each installment to every processing node is computed based on the computed load fractions and the buffer availability among all the processing nodes as given by (3.6) and (5.1).

The numerical example 6.1 illustrates the workings of the RASLD and RAPLD strategies with the various spanning tree construction algorithms. The parameters for this example are adapted from [30], so as to compare the results with the resource aware optimal load distribution with optimal sequencing (RAOLD-OS) strategy presented in [30], which is also a parallel distribution strategy.

Example 6.1**:**

Consider an arbitrary graph network G with nine nodes interconnected via 17 communication links, as shown in Fig. 6.3(a). Let the node processing speed parameters be $w_1 = 2.0$, $w_2 = 3.0$, $w_3 = 1.5$, $w_4 = 1.2$, $w_5 = 1.5$, $w_6 = 1.0$, $w_7 = 2.0$, $w_8 = 1.0$, $w_9 = 1.0$, and the link delay parameters be as marked in the Fig. 6.3(a). We let $T_{\text{cp}} = T_{\text{cm}} = 1$, and let the load $L = 100$ to originate at node C_9 in the given graph G.

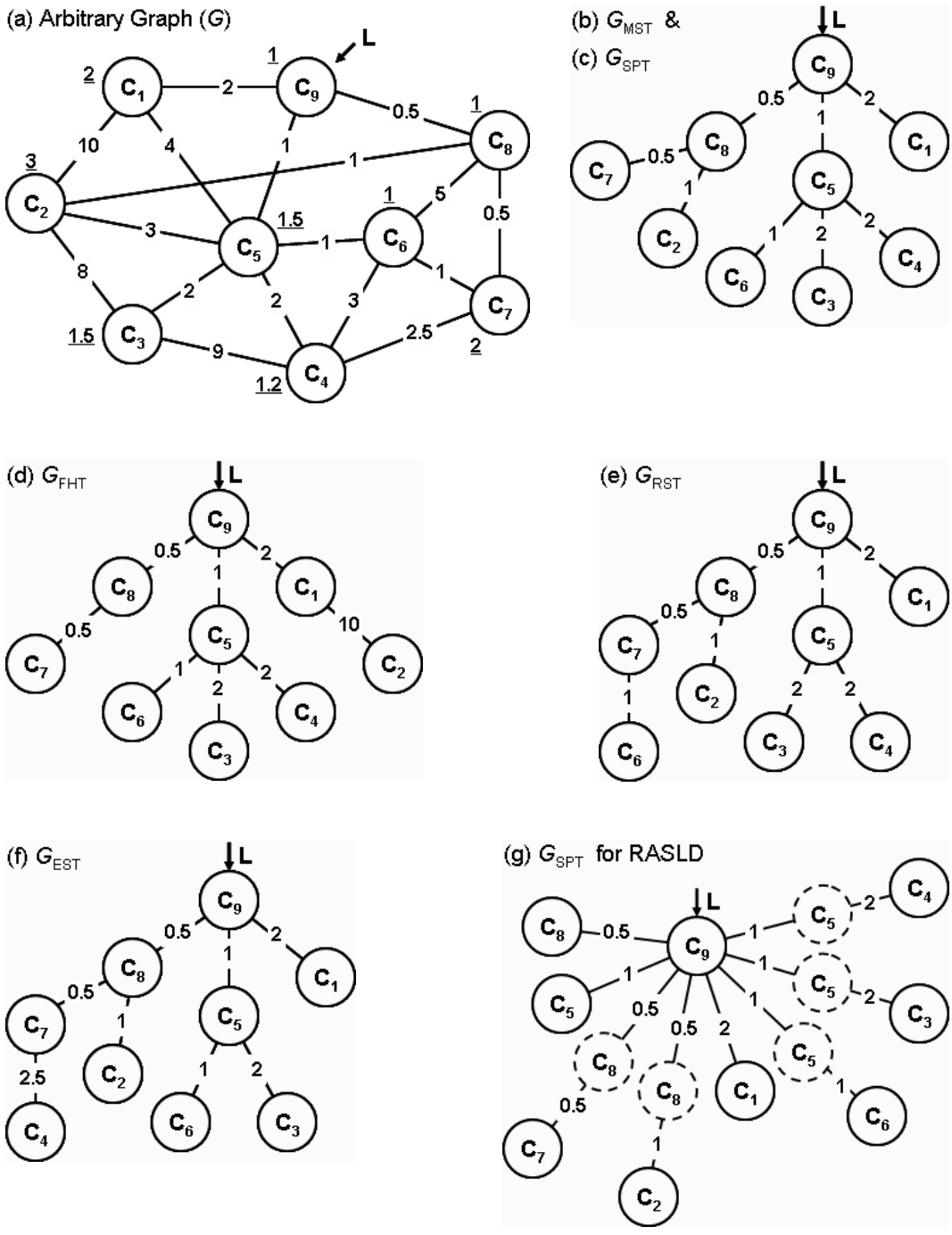

Figure 6.3: An arbitrary graph network and spanning trees and load distribution order on the spanning trees for Example 6.1. (Number on the links denote the link delay parameter ($z_{i,j}$) and the number near the nodes denote the processor speed parameter ($w_{i,j}$)). The load distribution order at every level on the spanning trees illustrated shall be from left to right; that is the order in which the link speeds ($\frac{1}{z_{i,j}}$) decrease. (a) An arbitrary graph network G with 9 nodes interconnected via 17 communication links; (b) Minimum spanning tree (G_{MST}); (c) Shortest path spanning tree (G_{SPT}); (d) Fewest hops spanning tree (G_{FHT}); (e) Robust spanning tree (G_{RST}); (f) Minimum network equivalence spanning tree (G_{EST}); and (g) G_{SPT} for Resource aware sequential load distribution (RASLD).

Table 6.1: Load distribution values with Resource aware sequential load distribution (RASLD), Resource aware parallel load distribution with minimum spanning tree (RAPLD(G_{MST})), shortest path spanning tree (RAPLD(G_{SPT})), fewest hops spanning tree (RAPLD(G_{FHT})), robust spanning tree (RAPLD(G_{RST})), and minimum network equivalence spanning tree (RAPLD(G_{EST})), and Resource aware optimal load distribution with optimal sequencing (RAOLD-OS) [30] strategies for Example 6.1.

Strategy	$\alpha_{9,1}$	$\alpha_{9,2}$	$\alpha_{\mathbf{9,3}}$	$\alpha_{9,4}$	$\alpha_{9,5}$	$\alpha_{9,6}$	$\alpha_{9,7}$	$\alpha_{9,8}$	$\alpha_{9,9}$	$T^*(\alpha^*)$
RASLD	2.01	2.68	0.30	0.11	12.07	1.34	6.04	30.18	45.27	45.27
RAPLD(G_{MST})	1.91	4.36	1.09	0.51	8.72	3.83	8.72	28.34	42.51	42.51
RAPLD(G_{SPT})	1.91	4.36	1.09	0.51	8.72	3.83	8.72	28.34	42.51	42.51
RAPLD(G_{FHT})	2.12	0.28	1.21	0.57	9.64	4.23	9.64	28.93	43.39	43.39
RAPLD(G_{RST})	2.05	3.10	2.34	1.10	7.75	6.20	7.75	27.89	41.84	41.84
RAPLD(G_{EST})	1.84	3.56	1.05	3.85	8.08	3.69	8.08	27.94	41.91	41.91
RAOLD-OS	2.58	4.90	1.47	0.69	6.87	5.15	9.79	24.48	44.07	44.07

The spanning trees (G_{MST}, G_{SPT}, G_{FHT}, G_{RST}, G_{EST}) generated by the MST, SPT, FHT, RST and EST construction strategies are shown in Fig. 6.3 (b), (c), (d), (e), and (f) respectively, and the spanning tree (G_{SPT}) for RASLD strategy is shown in Fig. 6.3(g). As seen from Fig. 6.3 the resultant spanning tree structures need not necessarily be different for each of the construction strategies. In this example, the MST and SPT generate identical spanning tree structures (G_{MST} and G_{SPT}). The load distributions and the optimal processing time obtained by the various strategies are tabulated in Table 6.1.

From the optimal processing time results in Table 6.1 it shall be noted that for this example RAPLD(G_{RST}) and not the RAPLD(G_{EST}) strategy provides the mini-

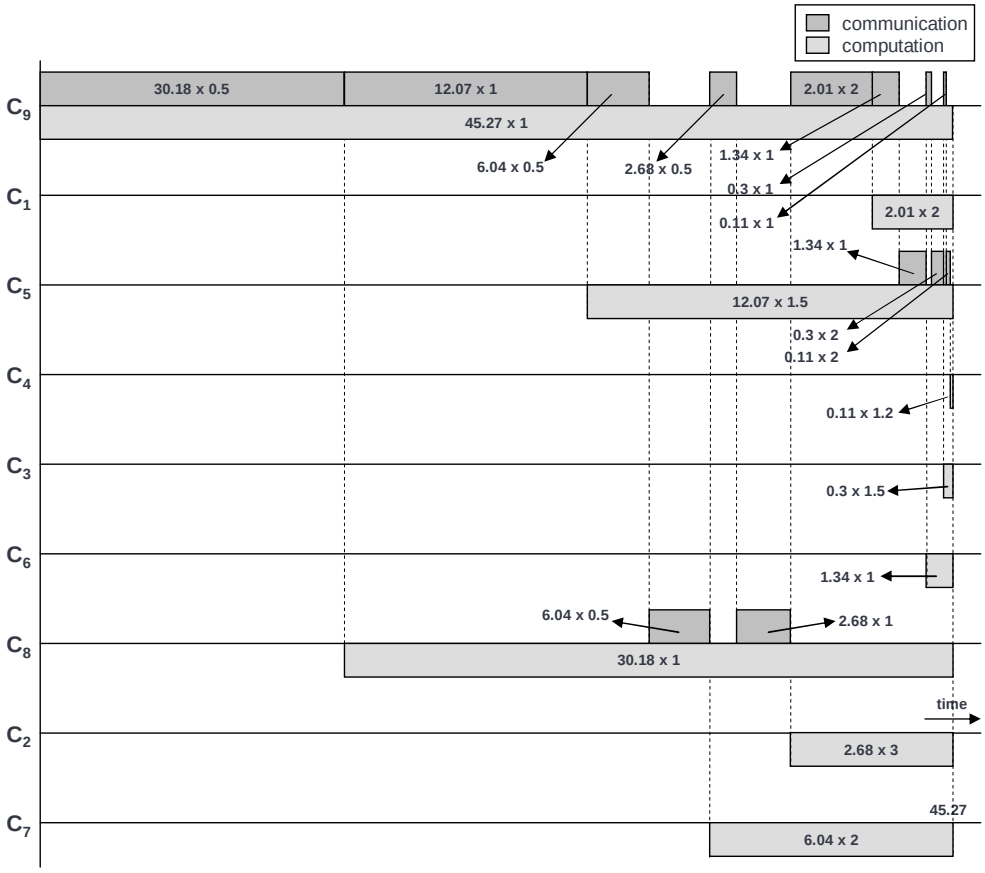

Figure 6.4: Timing diagram for the Resource aware sequential load distribution (RASLD) strategy (Example 6.1).

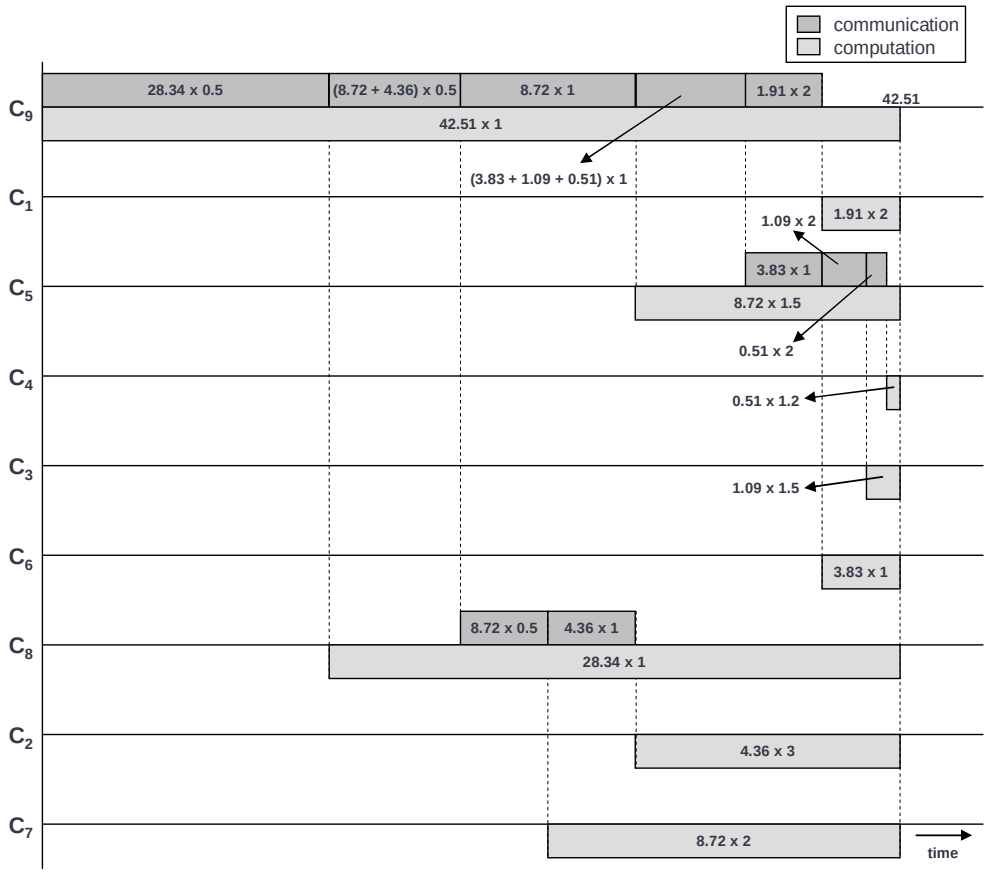

Figure 6.5: Timing diagram for the Resource aware parallel load distribution (RAPLD(G_{SPT})) strategy (Example 6.1).

mum value. It shall be noted that the EST algorithm do not provide global optimal tree. Though the EST algorithm takes into consideration both the processor and link speeds for spanning tree construction, the decision to add a node-link pair to the spanning tree is made at every step. Hence, the resultant spanning tree may not always provide the global optimal results.

The timing diagrams for the RASLD, RAPLD(G_{SPT}) and RAOLD-OS strategies are presented in Fig. 6.4, 6.5 and 6.6 for comparison purposes. These timing diagrams illustrate how and when the load fractions are to be communicated and

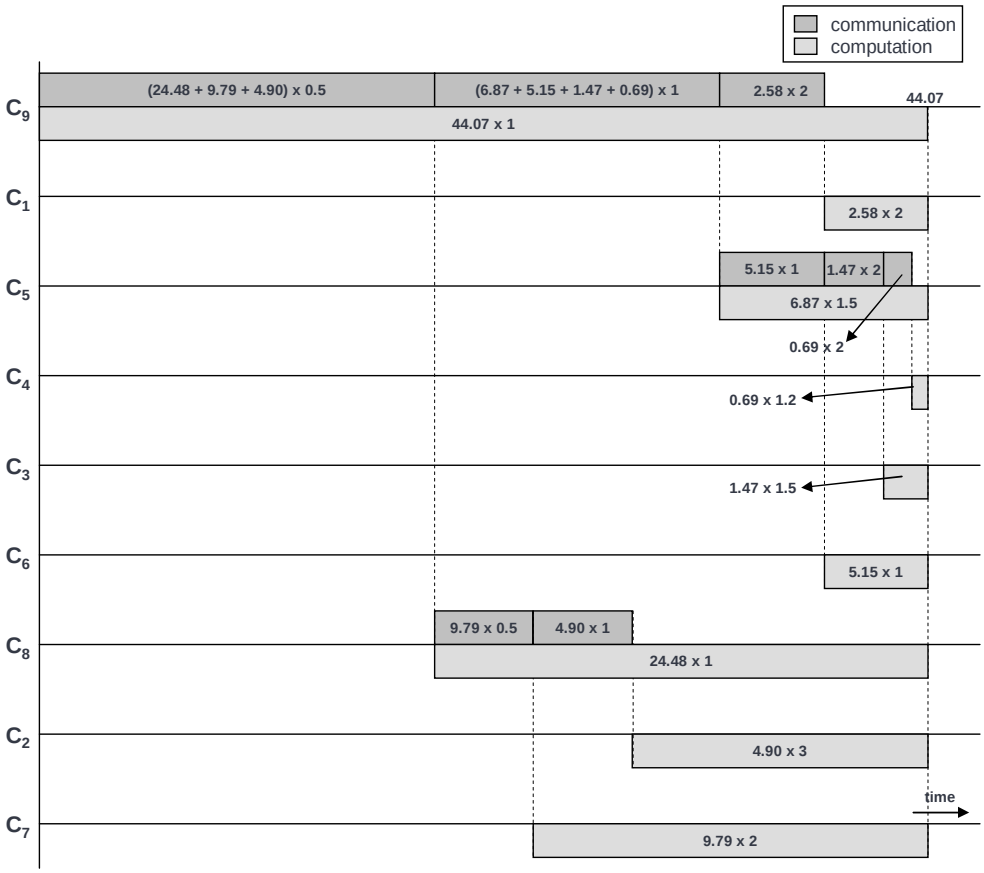

Figure 6.6: Timing diagram for the Resource aware optimal load distribution with optimal scheduling (RAOLD-OS) strategy [30] (Example 6.1).

computed by the nodes in the network. From the Fig. 6.6, it is seen that in the RAOLD-OS distribution strategy, the intermediate parent nodes receive the loads meant for them as well as their children, before start processing the load portions assigned to them. But, in our RAPLD strategy (Fig. 6.5), the intermediate parent nodes start processing the load portions assigned to them immediately upon receiving it from their parents. Hence, our RAPLD strategy always obtains minimum optimal processing time solution for a given spanning tree when compared with the RAOLD-OS strategy.

6.4 Performance Evaluation

In this section, we shall describe the simulation platform setup and analyze the performance of the proposed RASLD and RAPLD strategies with the various spanning tree construction strategies for several scenarios through extensive simulations. We also highlight and discuss all the important simulation results.

We now describe how the arbitrary graph networks and other required parameters for our performance evaluation are generated. The graph generation procedure is made to generate random graphs so as to reflect the real-life situations. We set the node C_0 in the network as the root node. The parameter P_{Link} denote the degree of connectivity, or *link density*. By varying the number of processing nodes and the link density parameter P_{Link} in our simulations, we generate various types of networks. This allows us to generate networks having very small number of nodes

with high connectivity as well as networks having large numbers of nodes with low or sparse connectivity, reflecting real-life scenarios.

In our study, we vary P_{Link} from 30% to 100% in steps of 10%, and generate various types of networks and for each type of network, we vary the number of processing nodes from 10 (very small-size network) to 300 (large-size network). This enables us to study the effects of network size scalability and network connectivity. It shall be noted that in order to guarantee the generated graph is a connected graph, the value of P_{Link} cannot be close to zero and when the value of P_{Link} is 100%, we have a fully connected network where in all the nodes are connected to each other by direct links. The speed parameters for the processing nodes and the links are chosen based on a uniform probability distribution in the range $[0.01, 3.34]$ for low, and $[6.67, 10.0]$ for high values. In all our studies, we let $L = 10^8$, $T_{\text{cm}} = T_{\text{cp}} = 1.0$, and vary the number of nodes in the network, processor and link speeds and P_{Link} values and analyze the performance. The simulation parameters and their respective ranges are given in Table 6.2.

6.4.1 Metrics of Interest

The metrics that are of significance to our study are the optimal processing time $(T^*(\alpha^*))$ and the network eccentricity, which we define as the distance from the root node to the farthest leaf node in the spanning tree. In addition to the optimal processing time, we consider network eccentricity in our study, since it provides an

Table 6.2: Simulation parameters and their range of values.

Parameter	**Range of values**
Number of nodes (N)	10, 100, 200, and 300
Link Density (P_{Link})	$30\% - 100\%$
Processing speed (Low \ High)	$(0.01 - 3.34) \setminus (6.67 - 10.0)$
Link speed (Low \ High)	$(0.01 - 3.34) \setminus (6.67 - 10.0)$
Communication Intensity Constant (T_{cm})	1
Computing Intensity Constant (T_{cp})	1
Load Size (L)	10^8

indication on how far the nodes are from the root node in a spanning tree. This metric also gives a measure of robustness of the network, since, farther the nodes are from the root node, more pronounced will be the effect of network disruptions on the performance because of data loss [59].

To compare the optimal processing time of various strategies we define a metric, normalized optimal processing time (δ) as

$$\delta = \frac{T^*(\alpha^*)_{RAPLD}}{T^*(\alpha^*)_{RASLD}} \tag{6.3}$$

where $T^*(\alpha^*)_{RAPLD}$ and $T^*(\alpha^*)_{RASLD}$ are the optimal processing time for RAPLD and RASLD strategies respectively.

We also define network eccentricity (ε) as

$$\varepsilon = Max(Number\ of\ hops\ from\ root\ node)\ \forall\ nodes\ \in\ G_{\text{Tree}} \tag{6.4}$$

which is the distance in number of hops from the root node to the farthest leaf node in the spanning tree.

The network eccentricity (ε) results are plotted in the Fig. 6.7. In Fig. 6.7, we denote the ε results for minimum spanning tree, shortest path spanning tree, fewest hops spanning tree, robust spanning tree and minimum network equivalence spanning tree as MST, SPT, FHT, RST, and EST respectively. Since the EST construction depends on both processor and link speeds, its ε value vary when the network has high and low processing speed nodes. Hence, they are plotted separately as EST(H) and EST(L) respectively.

The optimal processing time ($T^*(\alpha^*)$) results are plotted in Fig. 6.8 and 6.9 and normalized optimal processing time (δ) results are plotted in Fig. 6.10 and 6.11 for low and high link speed values respectively. In Fig. 6.8, 6.9, 6.10, and 6.11, we denote the results for RAPLD strategy with minimum spanning tree (RAPLD(G_{MST})), shortest path spanning tree (RAPLD(G_{SPT})), fewest hops spanning tree (RAPLD(G_{FHT})), robust spanning tree (RAPLD(G_{RST})), and minimum network equivalence spanning tree (RAPLD(G_{EST})) as MST, SPT, FHT, RST, and EST respectively and that of RASLD strategy (with SPT) as RASLD.

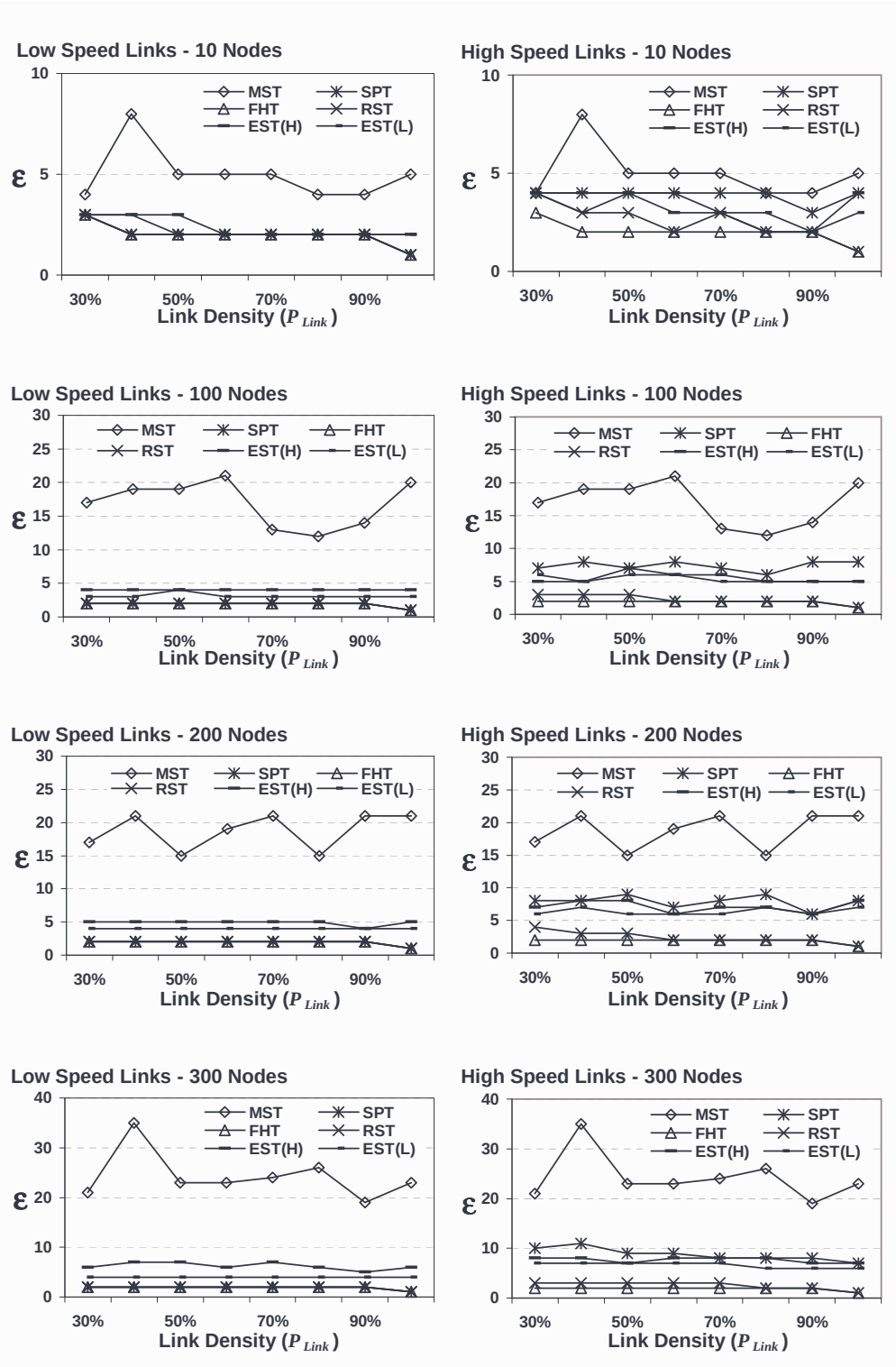

Figure 6.7: Network eccentricity (ε) results for minimum spanning tree (MST), shortest path spanning tree (SPT), fewest hops spanning tree (FHT), robust spanning tree (RST), and minimum network equivalence spanning tree (EST) construction strategies for $10, 100, 200$, and 300 nodes in a network with low and high speed links.

Note : Since, the ε value for EST vary when the network has high and low processing speed nodes, they are plotted separately as EST(H) and EST(L) respectively.

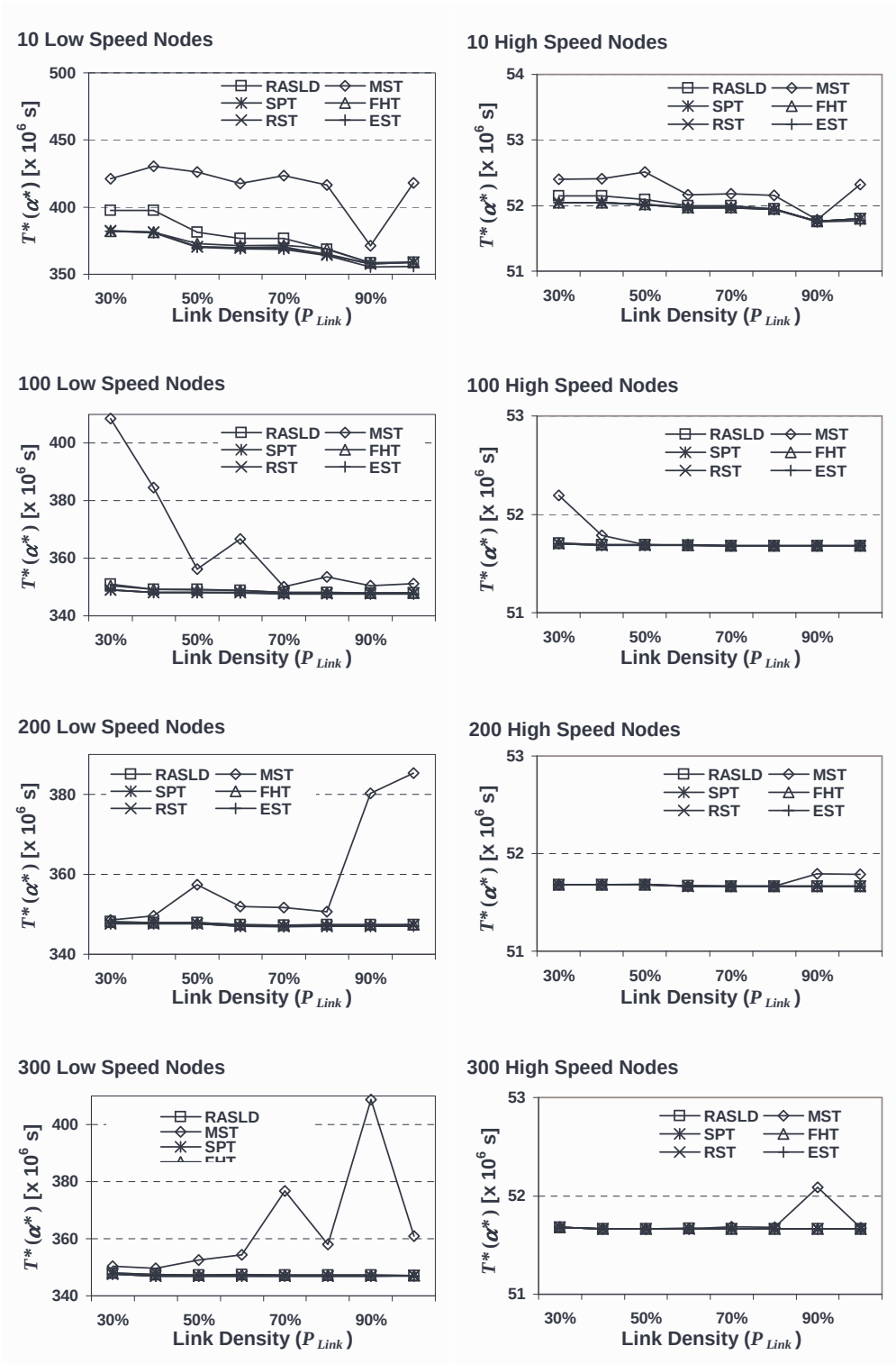

Figure 6.8: Optimal processing time ($T^*(\alpha^*)$) results for Resource aware sequential load distribution (RASLD) and Resource aware parallel load distribution (RAPLD) with minimum spanning tree (MST), shortest path spanning tree (SPT), fewest hops spanning tree (FHT), robust spanning tree (RST), and minimum network equivalence spanning tree (EST) construction strategies for $10, 100, 200$, and 300 nodes with low and high processing speeds in a network with low speed links.

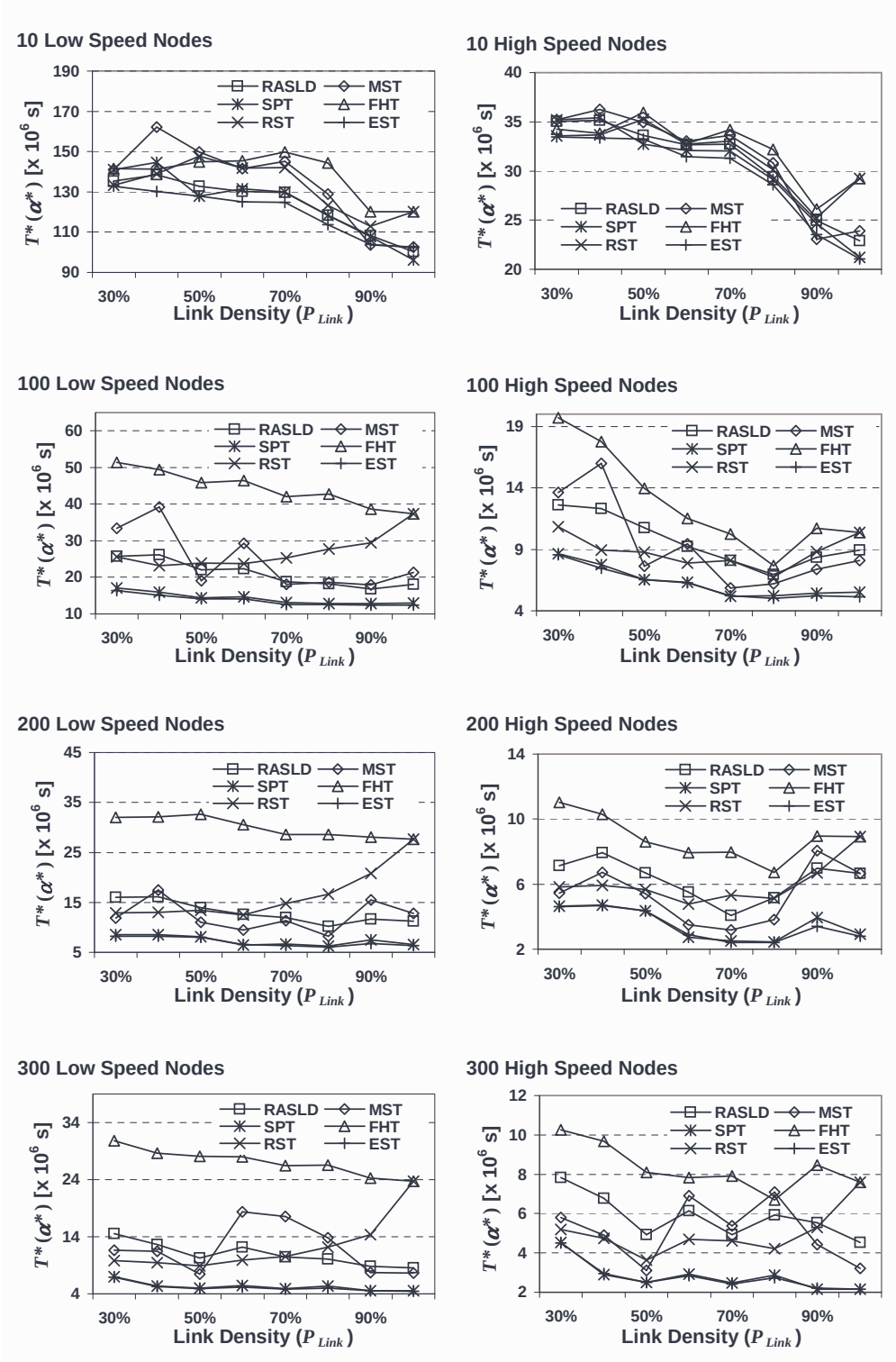

Figure 6.9: Optimal processing time ($T^*(\alpha^*)$) results for Resource aware sequential load distribution (RASLD) and Resource aware parallel load distribution with minimum spanning tree (MST), shortest path spanning tree (SPT), fewest hops spanning tree (FHT), robust spanning tree (RST), and minimum network equivalence spanning tree (EST) construction strategies for $10, 100, 200$, and 300 nodes with low and high processing speeds in a network with high speed links.

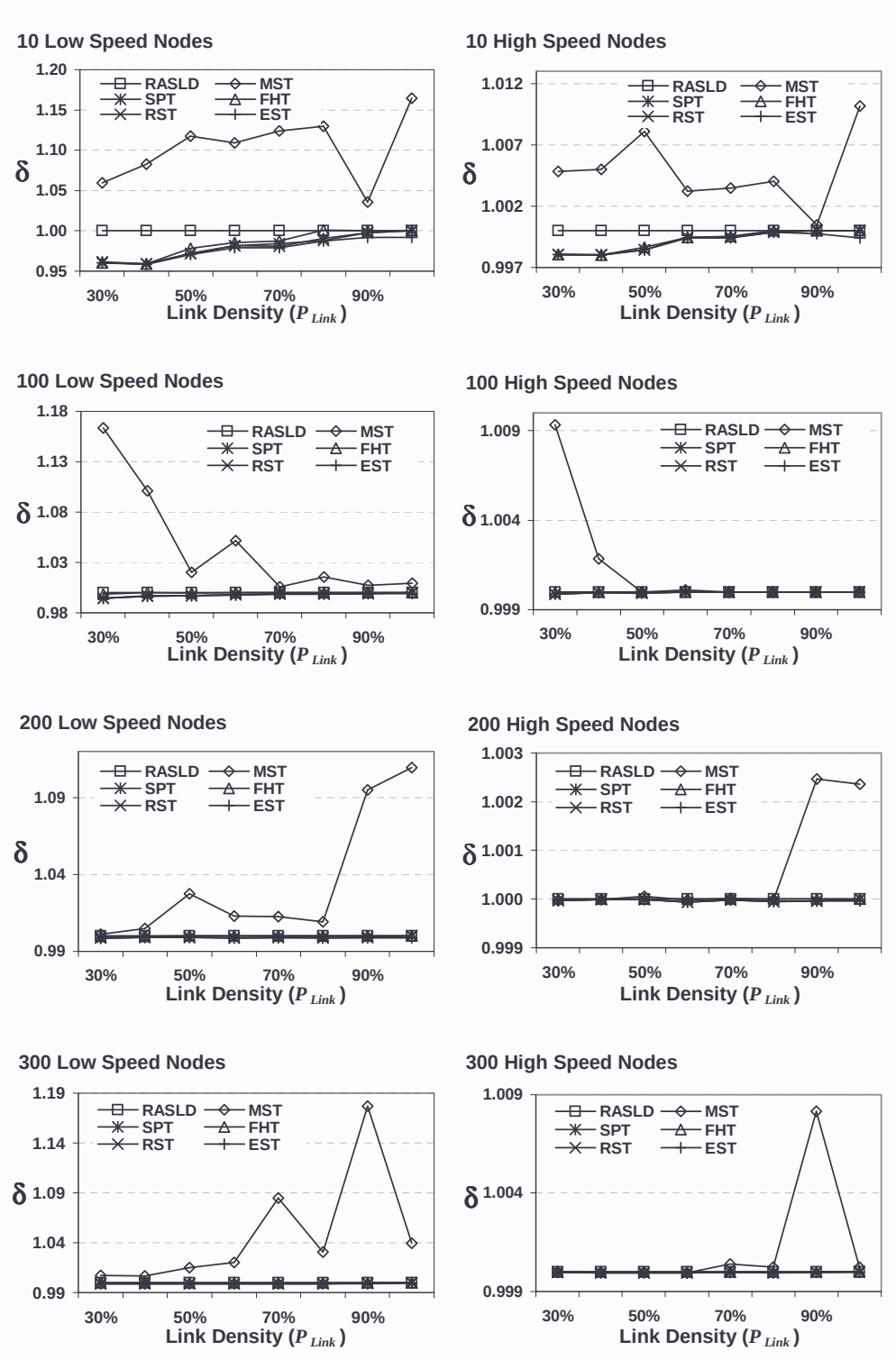

Figure 6.10: Normalized optimal processing time (δ) results for Resource aware sequential load distribution (RASLD) and Resource aware parallel load distribution with minimum spanning tree (MST), shortest path spanning tree (SPT), fewest hops spanning tree (FHT), robust spanning tree (RST), and minimum network equivalence spanning tree (EST) construction strategies for $10, 100, 200$, and 300 nodes with low and high processing speeds in a network with low speed links.

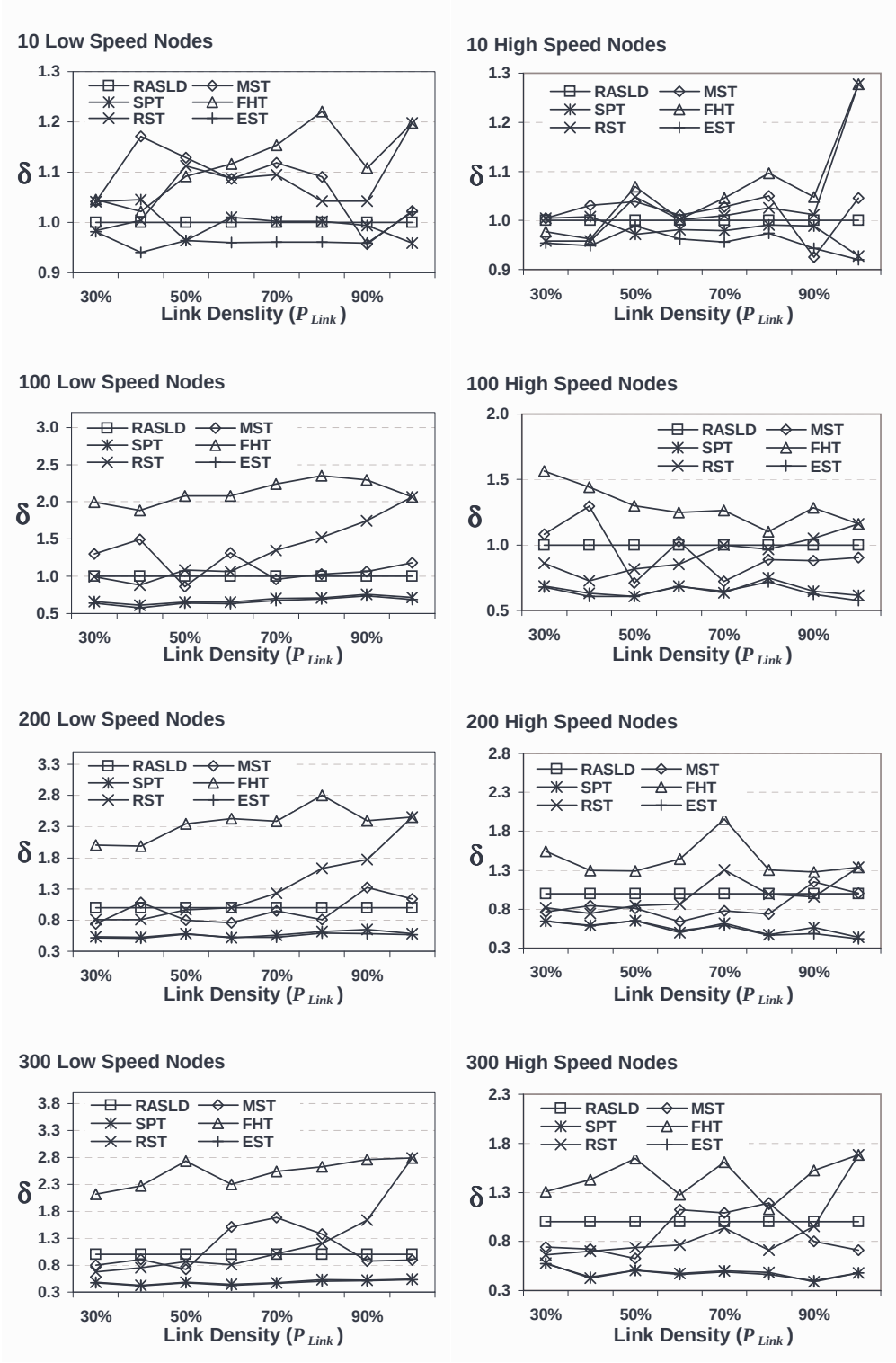

Figure 6.11: Normalized optimal processing time (δ) results for Resource aware sequential load distribution (RASLD) and Resource aware parallel load distribution with minimum spanning tree (MST), shortest path spanning tree (SPT), fewest hops spanning tree (FHT), robust spanning tree (RST), and minimum network equivalence spanning tree (EST) construction strategies for $10, 100, 200$, and 300 nodes with low and high processing speeds in a network with high speed links.

6.4.2 Effect of Network Scalability

We first study the effect of network scalability by comparing the performance of our proposed strategies for various processing node configurations for a given link density value. When the number of nodes in a network is increased while keeping the link density value constant, the number of links in the network proportionately increases with the number of nodes. Hence, it does not result in the increase of alternative routes available for the spanning tree algorithms.

From the Fig. 6.7, it is observed that the MST provides the upper bound and FHT provides the lower bound for the ε values and their values remain identical in both low and high link speed networks. For MST, the ε increases as the number of nodes in the network increases, whereas it almost remains unchanged in the case FHT. Both in the low and high link speed networks, the ε values for SPT, FHT, RST, and EST remain almost identical as the number of nodes in the network increases beyond 100, however the ε values for SPT and EST are generally higher for high link speed networks when compared with those for low link speed networks.

From the simulation results in Fig. 6.8 and 6.9, it is observed that the optimal processing time values decrease when the processing speed of node or the number of processing nodes in a network increases. It is also observed that the reduction in the processing time is not significant beyond a certain network size in terms of number of processing nodes for all strategies except RAPLD(G_{MST}). For example, in low link speed networks when the network size increases beyond 100

nodes and in high link speed networks when it increases beyond 200 nodes, no significant reduction in the processing time is observed for all strategies except RAPLD(G_{MST}). In low link speed networks, RAPLD(G_{MST}) and RAPLD(G_{EST}) are observed to produce upper and lower bounds respectively for the optimal processing time value, where as in the high link speed networks (except for the very sparse network with just 10 nodes) RAPLD(G_{FHT}) and RAPLD(G_{EST}) are observed to produce the upper and lower bounds. From the Fig. 6.10, it is observed that in the case of low link speed networks, beyond 100 nodes, the performance of all strategies except RAPLD(G_{MST}) are almost identical. From the Fig. 6.11, it is observed that beyond 10 nodes, RAPLD(G_{SPT}) and RAPLD(G_{EST}) strategies provide superior performance than all other strategies.

6.4.3 Effect of Network Connectivity

In order to analyze the effect of network connectivity, we compare the performance of the proposed distribution strategies by varying the link density value for a given number of processing nodes. When the link density value (P_{Link}) is increased for a given network, there are more links between processing nodes and hence there are more options available for the spanning tree algorithms.

From the Fig. 6.7, it is observed that the eccentricity values for SPT, RST, EST, and FHT tend to decrease slightly, and those for MST vary significantly as the P_{Link} increases in both low and high link speed networks.

Our simulations (Fig. 6.8, 6.9, 6.10, and 6.11) show that in low link speed networks with more than 100 nodes and in high speed networks with more than 200 nodes, irrespective of the processing speed of the nodes, there are only minor variations in the processing time value for RAPLD(G_{SPT}) and RAPLD(G_{EST}) construction strategies as P_{Link} increases. Also, the processing time values in the low link speed networks for all strategies except RAPLD(G_{MST}) are almost identical beyond 100 nodes. It is also observed that the optimal processing time value for RAPLD(G_{RST}) strategy is closer to those for RAPLD(G_{SPT}) strategy for low P_{Link} values and tends to move closer to the values for RAPLD(G_{FHT}) strategy as P_{Link} increases.

6.5 Complexity and Performance Comparison

Given an arbitrary graph $G = \langle N, E \rangle$, using Fibonacci heap, an MST and SPT could be constructed in $O(E + NlogN)$ steps. The complexity of BFS to construct FHT is $O(|E| + |N|)$. The complexity of constructing RST is $O(E^2)$. Assuming that there are m processing nodes in every sub-tree and that there are R sub-trees in every level, with a total number of Q levels in the entire tree network, the complexity to compute an equivalent processor speed value for the sub-tree and determining an optimal distribution is given by $O(RQ + RQlog(m))$. Hence, the construction of EST takes about $O(NERQ + NERQlog(m))$ steps. Since, $m \leq N$, $R \leq N$, $Q \leq N$, $N \leq NlogN$, and $RQ \leq RQlog(m)$, the total complexity of RAPLD(G_{MST}), RAPLD(G_{SPT}), and RAPLD(G_{FHT}) shall be approximated as

Table 6.3: Comparison of complexity and performance of Resource aware sequential load distribution (RASLD) and Resource aware parallel load distribution with minimum spanning tree (RAPLD(G_{MST})), shortest path spanning tree (RAPLD(G_{SPT})), fewest hops spanning tree (RAPLD(G_{FHT})), robust spanning tree (RAPLD(G_{RST})), and minimum network equivalence spanning tree (RAPLD(G_{EST})) strategies for divisible load scheduling.

Strategy	**Complexity**	**Performance**		
		Optimal processing time ($T^*(\alpha^*)$)		**Eccentricity (ε)**
		Low speed links	**High speed links**	
RASLD	$O(NlogN)$	Low	Medium	Medium
RAPLD(G_{MST})	$O(N^2logN)$	Highest	Medium	Highest
RAPLD(G_{SPT})	$O(N^2logN)$	Low	Lowest	Medium
RAPLD(G_{FHT})	$O(N^2logN)$	Low	Highest	Lowest
RAPLD(G_{RST})	$O(E^2 + N^2logN)$	Low	Medium	Low
RAPLD(G_{EST})	$O(E \cdot N^3logN)$	Low	Lowest	Medium

$O(N^2logN)$. Similarly, the total complexity of RAPLD(G_{RST}) and RAPLD(G_{EST}) shall be approximated as $O(E^2 + N^2logN)$ and $O(E \cdot N^3logN)$ respectively. For RASLD strategy, the SPT construction provides the necessary information for computing the optimal distribution. Hence, its complexity shall be approximated as $O(NlogN)$.

The complexity and optimal processing time performance comparisons are summarized in the Table 6.3. In general, it is seen that RAPLD(G_{EST}) provides the lowest processing time among all the strategies. However its complexity increases with number of nodes as well as number of links in a network. On the other hand, RAPLD(G_{SPT}) provides comparable performance while having far less complexity.

The time performance of RAPLD(G_{RST}) lies between that of RAPLD(G_{SPT}) and RAPLD(G_{FHT}) strategies, and it provides robustness when there are link failures. RAPLD(G_{MST}) seems to be the last option for divisible load scheduling in both low and high link speed networks. It is also seen that the eccentricity of FHT is the lowest and RST is comparable to that of FHT. The eccentricity of SPT and EST are slightly higher than FHT but much lower than that of MST.

In the case of low link speed networks, the optimal processing time performance of RASLD and RAPLD with all the spanning tree construction strategies are similar. Hence, the simpler RASLD is a better strategy for divisible load scheduling in low link speed networks.

In the case of high link speed networks, RAPLD(G_{EST}) and RAPLD(G_{SPT}) strategies seem to provide a better performance in terms of optimal processing time; and their trees are neither as "skinny" as MST nor as "fat" as FHT or RST. Hence, RAPLD(G_{SPT}) strategy is a better strategy for divisible load scheduling in high link speed networks.

However, the performance degradation of RAPLD(G_{RST}) is minimal for large network sizes as long as the link densities are moderate. Hence, if robustness against link failure is desired, then RAPLD(G_{RST}) strategy shall be considered in both low and high link speed networks.

Chapter 7

Conclusions and Future Work

In this chapter, we conclude comparing the performances of the load distribution strategies described in the earlier chapters in this thesis and provide some thoughts on further extensions.

The computational and data analysis capabilities offered by technologies are being extensively utilized for solving complex scientific problems. The expanding collaborations, data analysis requirements and increasing computational and networking capabilities led to the evolvement of Grid computing paradigm. Grid is always viewed as a repository of resources that can be availed by careful scheduling. Computational Grid systems (CGS) comprising of clusters of computers interconnected by high speed links are well suited for processing the unprecedented large volume divisible loads (data) that are being generated in various scientific domains like those in high energy nuclear physics experiments, bio-informatics etc.

Over the years, the divisible load theory (DLT) paradigm has been proven to be a very useful tool for handling large volume divisible loads in networked computing environments.

The contributions in this thesis has focused primarily on designing and analyzing polynomial-time complex scheduling strategies using the DLT paradigm for handling large volume divisible loads both within as well as across clusters in CGS. The work presented here is a first of its kind to consider real-time arrivals of divisible loads with firm deadlines and finite buffer constraints in a heterogeneous Grid environment.

7.1 Scheduling within Cluster Systems

We have proposed dynamic scheduling strategies for load distribution within clusters, utilizing the IBS algorithm proposed in the literature for off-line scheduling. Since our primary focus is to design strategies for scheduling loads, implicit to this problem are some real-life constraints such as availability of the nodes for processing, the amount of resources they can render, speeds with which the nodes and links can respond etc. Also, as in the case of any networked system, here too, we can follow a "pull-based" (processing nodes demand loads from the sources) or "push-based" (sources seek processing nodes and schedule the load for processing) approaches. In our strategies, we considered a pull-based strategy. We considered a real-life situation where in the sinks (processing nodes) have finite sized

buffer and hence it has to be shared in an optimal manner among the competing sources. Also, we assumed that every sink attempts to request loads from all the participating sources for processing.

Since the loads are submitted to the system at random times, the amount of buffer that a sink may render to each source also varies over time. The basis for the latter part of the above statement lies in the fact that the amount of buffer that can be rendered to the sources depends on the total load at that time. Thus, while designing algorithms for such dynamic situations, we considered tuning the IBS algorithm to handle random arrivals of the sources. Further, when the arrival rate is very high, the system may not be able to respond and hence may decide to drop some of the loads. While this is true with any practical system, the performance can be enhanced significantly by allowing buffer reclamation. This means that when the sinks allow certain amount of their buffer to be reclaimed by the scheduler, the number of loads that can be accommodated can be vastly improved.

We have also proposed distributed scheduling strategies for processing several time critical divisible loads (loads with deadline requirements). As in real-life situations, we have considered the dynamic arrival of loads, buffer capacity constraints at the sink nodes and the deadline requirement of the loads to be processed. We have proposed three schemes of RADIS and have rigorously analyzed and evaluated their performance. In our simulations for all the schemes, the number of sinks that participate in processing in an iteration are allowed to vary which is reflective

of real life situations. The impact of load sizes, load deadlines and buffer size variations are also captured in our simulation study.

In the RADIS schemes, the coordinator node performs the admissibility test, determines the loads to be processed and the sinks that participate in an iteration, computes the scheduling parameters required for the sink nodes for determining the schedule and communicates them to the sink nodes. The sink nodes perform the buffer estimation, estimate the amount of load fractions to be processed based on the scheduling parameters received from the coordinator node, determine the amount of load fractions to be requested from the source nodes based on the actual buffer availability at the start of an iteration and communicates the difference between the estimated and the actual amount of load processed in an iteration to the coordinator node. The sink nodes also request and process the amount of load fractions thus determined. In our schemes, the scheduling is done in a distributed manner (the load fractions are computed at the sink nodes) and only $(4 + r)$ variables (where r is the number of sources that are scheduled in an iteration) are communicated from the coordinator node to the sink nodes. Hence, the effectiveness of our schemes is more pronounced in larger systems. The proposed strategies are summarized in the Table 7.1. These strategies are well suited for scheduling within a cluster node in CGS.

Table 7.1: Summary of proposed scheduling strategies.

System	Assumptions	Algorithm
Scheduling within clusters:		
Nodes are geographically co-located within an organization and there are high speed links between them. There are dedicated buffers for processing divisible loads from other nodes in the network.	All nodes have front ends. Communication delay between nodes are negligible. Multi-port communication model. Dynamic arrival of loads. All sink nodes allow buffer reclamation.	***Divisible loads do not have deadline requirements:*** Dynamic IBS algorithm for time-invariant buffer environments. ***Divisible loads have deadline requirements:*** RADIS algorithm with Non-interleaved or EDF or Progressive scheduler for time-invariant buffer environments based on load arrival rates and deadline requirements of the loads.
Scheduling within clusters:		
Nodes are geographically co-located within an organization and there are high speed links between them. There are local tasks to be processed at the respective nodes in addition to processing the divisible loads from other nodes in the network. The buffer available at the nodes needs to be shared by the local tasks and the divisible loads from other nodes in the network. The time at which the local tasks arrive and their memory requirements are known a priori.	All nodes have front ends. Communication delay between nodes are negligible. Multi-port communication model. Dynamic arrival of loads. All sink nodes allow buffer reclamation.	***Divisible loads do not have deadline requirements:*** Dynamic IBS algorithm for predictable time-varying buffer environments. ***Divisible loads have deadline requirements:*** RADIS algorithm with Non-interleaved or EDF or Progressive scheduler for predictable time-varying buffer environments based on load arrival rates and deadline requirements of the loads.

Continued on Next Page...

Table 7.1: Summary of proposed scheduling strategies. – Continued

System	Assumptions	Algorithm
Scheduling within clusters:		
Nodes are geographically co-located within an organization and there are high speed links between them. There are local tasks to be processed at the respective nodes in addition to processing the divisible loads from other nodes in the network. The buffer available at the nodes needs to be shared by the local tasks and the divisible loads from other nodes in the network. The time at which the local tasks arrive and their memory requirements are not known.	All nodes have front ends. Communication delay between nodes are negligible. Multi-port communication model. Dynamic arrival of loads. All sink nodes allow buffer reclamation.	***Divisible loads do not have deadline requirements:*** Adaptive IBS algorithm. ***Divisible loads have deadline requirements:*** RADIS algorithm with Non-interleaved or EDF or Progressive scheduler for time-varying buffer environments based on load arrival rates and deadline requirements of the loads.
Scheduling across clusters:		
Clusters are geographically distributed across organizations. There are local tasks to be processed at clusters in addition to processing the divisible loads from other clusters. The buffer available at the clusters needs to be shared by the local tasks and the divisible loads from other clusters.	Divisible loads do not have deadline requirements. All nodes have front ends. Communication delay between clusters are not negligible. Uni-port communication model. Dynamic arrival of loads. At each cluster, there will be sufficient buffer space available to buffer/store the load portion allocated to it.	Resource aware sequential load distribution strategy (for networks with low link speeds). Resource aware parallel load distribution strategy with shortest path routing (RAPLD(G_{SPT})) (for networks with high link speeds). Resource aware parallel load distribution strategy with robust spanning tree routing (RAPLD(G_{RST})) (for large networks with both low and high speed links and moderate link densitites).

In summary, we infer and observe the following based on our work presented in this thesis.

- The DLT paradigm has been proven to be a valid tool for handling large scale computational loads on Cluster/Grid systems.

- Though our algorithms for scheduling are shown to provide best effort schedules, the under-estimation of buffer availability at the processing nodes shall enable all our schemes not to miss the deadlines for the accepted loads.

- All the proposed scheduling strategies are scalable, relevant in real-life situations and are shown to be useful under different scenarios.

- Although we have proposed a scheme for buffer estimation at sink nodes, it may be noted that the design of buffer estimation strategy is a topic in itself and other strategies such as the use of a fading memory could also be used with proposed strategies.

- For scheduling across clusters, there are lots of possible distribution strategies. The choice of the strategy depends on the adopted network communication model and speed of the links and processing nodes.

7.2 Scheduling across Cluster Systems

We have investigated the performance of sequential as well as parallel load distribution strategies in arbitrary network systems with communication delays between processing nodes (clusters). For the arbitrary graph network, a multi-level tree is constructed using spanning tree construction algorithms. Then, the multi-level tree is reduced to a single-level tree based on either the total communication delay along the path from the source node to the individual sink nodes (in the case of sequential distribution strategy) or recursively calculated equivalent processor speed

values (in the case of parallel distribution strategy). The optimal sequencing theorem proposed in the literature for single-level tree is then applied to derive solution for each of the strategies. Since, the shortest path spanning tree provides the least delay path between the root and the processing nodes, with optimal sequencing it provides an optimal performance for sequential distribution.

The strategies proposed are well suited for dynamic scenarios, where in there are dynamic load arrivals, finite buffer capacity constraints at the processing nodes, some of the nodes may or may not participate in processing of loads (based on their buffer availability), and some of the nodes may or may not communicate the loads to their children. Our proposed strategies considered load from one source at a time, so that admissibility testing could be performed for time critical loads. Our strategies assume that the nodes participating in computation for an accepted load, shall not leave the system until the processing is completed for the load portions assigned to it as well as the sub-tree for which it is the parent, else the guarantees given to the loads while admitting may not be fulfilled.

The performance of all the proposed strategies have been evaluated for wide range of arbitrary graphs with varying connectivity and node densities. Our simulations study shows that the simpler RASLD strategy shall be employed in case of systems with low speed links irrespective of the processing speed of the nodes. In the case of systems with high speed links, the RAPLD strategy is seen to provide better time performance. In such systems, the RAPLD(G_{SPT}) strategy provides a better trade-off between time complexity and performance while the

RAPLD(G_{RST}) strategy renders better trade-off between performance and robustness. RAPLD(G_{EST}) strategy, on the other hand, is seen to deliver the best performance, however with large time complexities. The proposed strategies are summarized in the Table 7.1.

7.3 Future Work

In the schemes presented for scheduling time critical loads, admissibility testing is being performed by the coordinator node. Hence, there is a chance for single point failure. However, one could implement a distributed approach using leader election like algorithms [54] to make it more fault tolerant.

We have considered uni-port communication model and the link delays alone while proposing strategies for scheduling across clusters. However, in Grid systems with high speed links and nodes with multi-core processors, concurrent communication in different links is certainly a viable model for handling large scale computational loads such as the one addressed in this thesis. Hence, strategies adopting concurrent communications in different links, and absorbing link and processor availability factors into the cost function for overall processing time minimization shall be explored.

Also, we have presented solutions that consider load from one source at a time. However, since, our strategies reduce the multi-level tree to a single-level tree, the schemes proposed in the literature for concurrently scheduling loads from multiple

sources on single-level tree networks, like those presented by Moges et al [64] or Xiaolin et al [65] could be considered and extended for handling time critical loads and other real life scenarios like buffer capacity variations at the processing nodes.

In some of the computational Grids, the number of interconnected clusters and also the total number of nodes in the system could be less, in which case, instead of multi-level hierarchical strategies, an all-to-all strategy could be designed considering both the communication delays between nodes and buffer capacity variations at the nodes to handle time critical loads.

In this thesis, efficient strategies have been designed and their performance have been analyzed with simulation studies. However, while applying these strategies onto a real Grid system, several other factors such as monetary cost charged for the utilization of the resources, storage requirements etc shall also be considered.

Scheduling approaches with emphasis on fault tolerance considering random node and link failures in a Grid system is also a green field for future research activities.

Bibliography

[1] Foster, I., "The Grid: A New Infrastructure for 21st Century Science," *Physics Today*, vol. 55, no. 2, pp. 42-47, Feb 2002.

[2] Foster, I., and Kesselman, C., "The Grid: Blueprint for a New Computing Infrastructure," *Morgan Kaufman*, 1999.

[3] Foster, I., Kesselman, C., and Tuecke, S., "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Int'l Journal of Supercomputer Applications*, vol. 15, no. 3, pp. 200-222, 2001.

[4] Johnston, W., Gannon, D., and Nitzberg, B., "Grids as production computing environments: The engineering aspects of NASAs information power grid," 8^{th} *IEEE Int'l Symposium on High Performance Distributed Computing*, pp. 197-204, Aug 1999.

[5] "NSF Tera-Grid," `http://www.teraGrid.org/`.

[6] Vieira, G.E., Hermann, J.W., and Lin, E., "Rescheduling manufacturing systems: a framework of strategies, policies and methods," *Journal of Scheduling*,

vol. 6, no. 1, pp. 36-92, Jan 2003.

[7] Hoschek, W., Jaen-Martinez, J., Samar, A., Stockinger, H., and Stockinger, K., "Data management in an international data Grid project," 1^{st} *IEEE/ACM Int'l Workshop on Grid Computing (Grid2000)*, pp. 77-90, Dec 2000.

[8] "W3C: World Wide Web Consortium," `http://www.w3.org/`.

[9] Baker, M., Buyya, R., and Laforenza, D., "Grids and Grid technologies for wide-area distributed computing," *Software - Practice and Experience*, vol. 32, no. 15, pp. 1437-1466, Dec 2002.

[10] Ming, W., and Xian-He, S., "Memory Conscious Task Partition and Scheduling in Grid Environments," *Proc. of* 5^{th} *IEEE/ACM Int'l Workshop on Grid Computing*, pp. 138-145, Nov 2004.

[11] Korkhov, V.V., Moscicki, J.T., and Krzhizhanovskaya, V.V., "Dynamic workload balancing of parallel applications with user-level scheduling on the Grid," to appear in *Future Generation Computer Systems*, Elsevier B.V., 2008.

[12] Kim, S., and Weissman, J.B., "A Genetic Algorithm Based Approach for Scheduling Decomposable Data Grid Applications," *Int'l Conf. on Parallel Processing (ICPP) 2004*, vol. 1, pp. 406-413, Aug 2004.

[13] Ruchir, S., Bhardwaj, V., and Misra, M., "On the Design of Adaptive and Decentralized Load Balancing Algorithms with Load Estimation for Computational Grid Environments," *IEEE Trans. on Parallel and Distributed Systems*, vol. 18, no. 12, pp. 1675-1686, Dec 2007.

[14] Foster, I., "Globus Toolkit Version 4: Software for Service-Oriented Systems," *Journal of Computer Science and Technology*, vol. 21, no. 4, pp. 513-520, Jul 2006.

[15] "The Dutch Distributed Advanced School for Computing and Imaging (ASCI) Supercomputer 2 (DAS-2)," `http://www.cs.vu.nl/das2/`.

[16] Veeravalli, B., Ghose, D., Mani, V., and Robertazzi, T.G., "Scheduling Divisible Loads in Parallel and Distributed Systems," *IEEE Computer Society Press*, Sep 1996.

[17] Cheng, Y.C., and Robertazzi, T.G., "Distributed Computation for a Tree Network with Communication Delays," *IEEE Trans. on Aerospace and Electronic Systems*, vol. 26, no. 3, pp. 511-516, May 1990.

[18] Blazewicz, J., and Drozdowski, M., "Scheduling Divisible Jobs on Hypercubes," *Parallel Computing* , vol. 21, pp. 1945-1956, 1995.

[19] Kim, H.J., Jee, G.I., and Lee, J.G., "Optimal Load Distribution for Tree Network Processors," *IEEE Trans. on Aerospace and Electronic Systems*, vol. 32, no. 2, pp. 607-612, Apr 1996.

[20] Drozdowski, M., and Glazek, W., "Scheduling Divisible Loads in a Three-Dimensional Mesh of Processors," *Parallel Computing* , vol. 25, no. 4, pp. 381-404, Apr 1999.

[21] Veeravalli, B., Ghose, D., and Mani, V., "Multi-installment Load Distribution in Tree Networks with Delays," *IEEE Trans. on Aerospace & Electronic Systems* , vol. 31, no. 2, pp. 555-567, Apr 1995.

[22] Kim, H.J., "A Novel Optimal Load Distribution Algorithm for Divisible Loads," *Special issue of Cluster Computing on Divisible Load Scheduling*, vol. 6, no. 1, pp. 41-46, Jan 2003.

[23] Juim, T.H., Kim, H.J., and Robertazzi, T.G., "Scalable scheduling in parallel processors," *Proc. of Conf. on Information Sciences and Systems*, Mar 2002.

[24] Robertazzi, T.G., "Ten Reasons to Use Divisible Load Theory," *Computer*, vol. 36, no. 5, pp. 63-68, May 2003.

[25] Veeravalli, B., Ghose, D., and Robertazzi, T.G., "Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems," *Special issue of Cluster Computing on Divisible Load Scheduling*, vol. 6, no. 1, pp. 7-18, Jan 2003.

[26] Agrawal, R., and Jagadish, H.V., "Partitioning Techniques for Large-Grained Parallelism," *IEEE Trans. On Computers*, vol. 37, no. 12, pp. 1627-1634, Dec 1988.

[27] Cheng, Y.C., and Robertazzi, T.G., "Distributed Computation with Communication Delays," *IEEE Trans. on Aerospace and Electronic Systems*, vol. 24, no. 6, pp. 700-712, Nov 1988.

[28] Haddad, E., "Real-time optimization of distributed load balancing," *Proc. of the 2^{nd} Workshop on Parallel and Distributed Real-Time Systems*, pp. 52-57, Apr 1994.

[29] Marchal, L., Yang, Y., Casanova, H., and Robert, Y., "A Realistic Network/Application Model for Scheduling Divisible Loads on Large-Scale Platforms," *Int'l Parallel and Distributed Processing Symposium (IPDPS) 2005*, pp. 48b-48b, Apr 2005.

[30] Yao, J., and Veeravalli, B., "Design and Performance Analysis of Divisible Load Scheduling Strategies on Arbitrary Graphs," *Cluster Computing*, vol. 7, no. 2, pp. 841-865, 2004.

[31] Lin, X., Lu, Y., Deogun, J., and Godard, S., "Real-Time Divisible Load Scheduing for Cluster Computing," 13^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07)*, pp. 303-314, Apr 2007.

[32] Ghose, D., Kim, H.J., and Kim, T.H., "Adaptive Divisible Load Scheduling Strategies for Workstation Clusters with Unknown Network Resources," *IEEE Trans. on Parallel and Distributed Systems*, vol. 16, no. 10, pp. 897-907, Oct 2005.

[33] Sohn, J., and Robertazzi, T.G., "An Optimal Load Sharing Strategy for Divisible Jobs with Time-varying Processor Speeds," *IEEE Trans. on Aerospace and Electronic Systems*, vol. 34, no.3, pp. 907923, Jul 1998.

[34] Li, X., Veeravalli, B., and Ko, C.C., "Divisible Load Scheduling on Single Level Tree Networks with Buffer Constraints," *IEEE Trans. on Aerospace and Electronic Systems*, vol. 36, no. 4, pp. 1298-1308, Oct 2000.

[35] Veeravalli, B., and Barlas, G., "Scheduling Divisible Loads with Processor Release Times and Finite Size Buffer Capacity Constraints," *Special issue of Cluster Computing on Divisible Load Scheduling*, vol. 6, no. 1, pp. 63-74, Jan 2003.

[36] Ghose, D., "A Feedback Strategy for Load Allocation in Workstation Clusters with Unknown Network Resource Capabilities using the DLT Paradigm," *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, vol. 1, pp. 425-428, Jun 2002.

[37] Rosenberg, A.L., "Sharing Partitionable Workload in Heterogeneous NOWs: Greedier is Not Better," *Proc. of IEEE Int'l Conf. on Cluster Computing*, pp. 124-131, 2001.

[38] Sohn, J., Robertazzi, T.G., and Luryi, S., "Optimizing Computing Costs using Divisible Load Analysis," *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 3, pp. 225-234, Mar 1998.

[39] Beaumont, O., Casanova, H., Legrand, A., Robert, Y., and Yang, Y., "Scheduling divisible loads on star and tree networks: results and open problems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 16, no. 3, pp. 207-218, Mar 2005.

[40] Ghose, D., and Kim, H.J., "Computing BLAS level-2 operations on workstation clusters using the divisible load paradigm," *Mathematical and Computer Modelling*, vol. 41, no. 1, pp. 49-70, Jan 2005.

[41] Chan, S.K., Veeravalli, B., and Ghose, D., "Large Matrix-vector Products on Distributed Bus Networks with Communication Delays using the Divisible Load Paradigm: Performance Analysis and Simulation," *Mathematics and Computers in Simulation*, vol. 58, pp. 71-79, 2001.

[42] Blazewicz, J., Drozdowski, M., and Markiewicz, M., "Divisible Task Scheduling - Concept and Verification," *Parallel Computing*, vol. 25, no. 1, pp. 87-98, Jan 1999.

[43] Drozdowski, M., and Wolniewicz, P., "Experiments with Scheduling Divisible Tasks in Clusters of Workstations," *EURO-Par-2000, Lecture Notes in Computer Science, Springer-Verlag*, no. 1900, pp. 311-319, 2000.

[44] Blazewicz, J., Ecker, K., Plateau, B., and Trystram, D., "Hand-book on Parallel and Distributed Processing," *Springer*, 2000.

[45] Li, P., Veeravalli, B., and Kassim, A.A., "Design and Implementation of Parallel Video Encoding Strategies using Divisible Load Analysis," *IEEE Trans. on Circuits and Systems for Video technology (CSVT)*, vol. 15, no. 9, pp. 1098-1112, Sep 2005.

[46] Bharadwaj, V., and Ranganath, S., "Theoretical and Experimental Study on Large Size Image Processing Applications using Divisible Load Paradigm on

Distributed Bus Networks," *Image and Vision Computing, Elsevier Publishers, USA*, vol. 20, issues 13-14, pp. 917-936, Dec 2002.

[47] Li, X., Veeravalli, B., and Ko, C.C., "Experimental study on processing divisible loads for large size image processing applications using PVM clusters," *Int'l Journal of Computers and Applications*, Jul 2001.

[48] Wong, H.M., and Bharadwaj, V., "Aligning Biological Sequences on Distributed Bus Networks: A Divisible Load Scheduling Approach," *IEEE Trans. on Information Technology in BioMedicine*, vol. 9, no. 4, pp. 489-501, Dec 2005.

[49] Charcranoon, S., Robertazzi, T.G., and Luryi, S., "Parallel Processor Configuration Design with Processing/Transmission Costs," *IEEE Trans. on Computers*, vol. 40, no. 9, pp. 987-991, Sep 2000.

[50] Drozdowski, M., and Wolniewicz, P., "Performance Limits of Divisible Load Processing in Systems with Limited Communication Buffers," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, pp. 960-973, 2004.

[51] Yu, D., and Robertazzi, T.G., "Divisible Load Scheduling for Grid Computing," *IASTED International Conference on Parallel and Distributed Computing and Systems 2003*, pp. 1-6, Nov 2003.

[52] Aaron, E.D., Lucas, C., and Wu-chun, F., "The Design, Implementation, and Evalulation of mpiBLAST," *Proc. of* 4^{th} *Int'l Conf. on Linux Clusters: The HPC Revolution 2003*, Jun 2003.

[53] Naotaka, Y., Osamu, T., and Satoshi, S., "Parallel and Distributed Astronomical Data Analysis on Grid Datafarm grid," *Proc. of* 5^{th} *IEEE/ACM Int'l Workshop on Grid Computing*, pp. 461-466, Nov 2004.

[54] Attiya, H., and Welch, J., *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, USA: McGraw Hill Publishers, 1998.

[55] Piriyakumar, D.A.L., and Murthy, C.S.R., "Distributed computation for a hypercube network of sensor-driven processors with communication delays including setup time," *IEEE Trans. on Systems, Man, and Cybernetic Part A*, vol. 28, no. 2, pp. 245-251, Mar 1998.

[56] Wong, H.M., Yu, D., Veeravalli, B., and Robertazzi, T.G., "Data Intensive Grid Scheduling: Multiple Sources with Capacity Constraints," *IASTED Int'l Conf. on Parallel and Distributed Computing and Systems 2003*, pp. 7-11, Nov 2003.

[57] Wong, H.M., Bharadwaj, V., and Barlas, G., "Design and Performance Evaluation of Load Distribution Strategies for Multiple Divisible Loads on Heterogeneous Linear Daisy Chain Networks", *Journal of Parallel and Distributed Computing*, vol. 65, no. 12, pp. 1558-1577, Dec 2005.

[58] Jingxi, J., Veeravalli, B., and Ghose, D., "Adaptive Load Distribution Strategies for Divisible Load Processing on Resource Unaware Multilevel Tree Networks," *IEEE Trans. on Computers*, vol. 56, no. 7, pp. 999-1005, Jul 2007.

[59] England, D., Veeravalli, B., and Weissman, J.B., "A Robust Spanning Tree Topology for Data Collection and Dissemination in Distributed Environments," *IEEE Trans. on Parallel and Distributed Systems*, vol. 18, no. 5, pp. 608-620, May 2007.

[60] Byrnes, P., and Miller, L.A., "Divisible load scheduling in distributed computing environments: complexity and algorithms," *Technical Report MN-ISYE-TR-06-006*, University of Minnesota Graduate Program in Industrial and Systems Eng., 2006.

[61] Stankovic, J.A., Spuri, M., Ramamritham K., and Buttazzo G.C., "Deadline scheduling for real-time systems : EDF and related algorithms," *Springer*, 1998.

[62] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, "Introduction to Algorithms, Second Edition," *MIT Press and McGraw-Hill,* 2001.

[63] Knuth, D.E., "The Art of Computer Programming," vol. 1, *Addison-Wesley*, Boston, 1997.

[64] Moges, M.A., Yu, D., and Robertazzi, T.G., "Grid Scheduling Divisible Loads from Multiple Sources via Linear Programming," *Int'l Conf. on Parallel and Distributed Computing and Systems (PDCS) 2004*, pp. 423-428, Nov 2004.

[65] Li, X., and Veeravalli, B., "A Processor-set Partitioning and Data Distribution Algorithm for Handling Divisible Loads from Multiple Sites in Single-Level Tree Networks," to appear in *Cluster Computing*.

Author's Publications

[1] Sivakumar, V., Bharadwaj, V., Yu, D., and Robertazzi, T.G., "Design and Analysis of a Dynamic Scheduling Strategy with Resource Estimation for Large-Scale Grid Systems," in *Proc. of* 5^{th} *IEEE/ACM Int'l Workshop on Grid Computing (Grid 2004)*, Pittsburgh, USA, pp. 163-170, Nov 2004.

[2] Sivakumar, V., Bharadwaj, V., and Robertazzi, T.G., "Resource Aware Distributed Scheduling Strategies for Large-Scale Computational Grid Systems," in *IEEE Trans. on Parallel and Distributed Systems*, vol. 18, no. 10, pp. 1450-61, Oct 2007.

[3] Sivakumar, V., Bharadwaj, V., and Jingxi, J., "Spanning Tree Routing Strategies for Divisible Load Scheduling on Arbitrary Graphs - a Comparative Performance Analysis," to appear in *IEEE Int'l Conf. on High Performance Computing (HiPC 2009)*, Kochi, India, Dec 2009.

[4] Sivakumar, V., and Bharadwaj, V., "Design and Analysis of Distribution Strategies for Divisible Loads on interconnected clusters of Large-Scale Computational Grid Systems," submitted to *Journal of Parallel and Distributed*

Computing.