COST-BASED PARTITIONING FOR DISTRIBUTED SIMULATION OF HIERARCHICAL MODULAR DEVS MODELS

by

Sunwoo Park

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2003

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Sunwoo Park

entitled Cost-based Partitioning For Distributed Simulation of Hierarchical Modular DEVS Model

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Bernard P. Zeigler, Ph.D.	May 04/03 Date
Kevin M. McNeill, Ph.D.	May 6/03 Date
Salim Hariri, Ph.D.	05/06/03 Date
D.P. Guertin, Ph.D.	05/06/03 Date
George L. Ball, Ph.D.	05/07/03 Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Bernard P. Zeigler, Ph.D.	May 04/03 Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from the dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: 

ACKNOWLEDGMENTS

I would like to express my gratitude to Professor Bernard P. Zeigler, my advisor, for his invaluable guidance, assistance, and support during my research period. I would also like to thank to Professor Salim Hariri, Professor Kevin McNeill, Professor George Ball, and Phillip Guertin, who served on my committee, for their suggestions and help in completing my degree requirements. I appreciate all their time and effort.

There are many other people who deserve special thanks. My former and current colleagues, Dr. HyupJae Cho, Dr. Yougkwan Cho, Dr. Jongsik Lee, Dr. Jongkeun Lee, Mr. James Nutaro, Mr. Chungman Seo, and Mr. Saehoon Cheon. Without their suggestions and support, I could not have finished my degree program. I appreciate all their friendship and assistance. Furthermore, I could not forget to thank to all members of Arizona Center for Integrated Modeling and Simulation.

My sincere appreciation goes to my father and mother who gave me the meaning of life.

To my parents

TABLE OF CONTENTS

TABLE OF CONTENTS - Continued

TABLE OF CONTENTS - Continued

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS - Continued

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT

The main objective of this research is to design and implement a class of generic partitioning algorithms for hierarchical, modular Discrete Event Specification System (DEVS) models for distributed simulation. To attain the goal of this dissertation, a set of partitioning algorithms is designed using the cost analysis methodology.

For more than a decade, abundant research has been conducted to develop partitioning algorithms that can find optimal, or reasonably acceptable, solutions for various partitioning problems. These employ methods such as simulated annealing, random partitioning, heuristic partitioning, and hierarchical clustering. In this dissertation, a new Generic Model Partitioning (GMP) algorithm for hierarchical modular DEVS models is proposed for distributed simulation.

The GMP algorithm decomposes a given hierarchical model into a set of partition blocks and provides reasonable solutions for distinct partitioning problems based on a cost analysis methodology. It also minimizes model decomposition during the partitioning process and guarantees incremental quality of partitioning (QoP) improvements until a best partitioning is attained. A series of cost measures for cost generation, cost evaluation, and cost aggregation are introduced. Since a cost measure is a parametric method, subject to certain axioms, the proposed algorithm is generic and applicable any family of models provided there is a way to manipulate the appropriate cost information.

A class of advanced algorithms derived from the GMP algorithm is also presented to tackle sophisticated issues associated with various distributed system configurations.

CHAPTER 1. INTRODUCTION

1.1 Motivation and Goals

For more than a decade, research has been conducted to develop Modeling and Simulation (M&S) frameworks over various distributed network infrastructures. As a result, numerous frameworks have been implemented to achieve a certain level of satisfaction with respect to *connectivity*, *speedup*, and *resource utilization*. By connecting numbers of computers that are loosely or tightly coupled in distributed network infrastructure and sharing resource information between M&S entities, desired M&S activities are conducted faster and more efficiently as compared the same activities on a standalone system. A certain level of interoperability and collaboration between M&S entities is also achieved in those frameworks.

In distributed simulation, there are many issues effecting simulation performance. Time synchronization and communication overhead are some of the major issues. Those issues are mainly related to simulation entities (e.g., logical simulator, coordinator, and activator) that are involved in the actual simulation process. By synchronizing time information with desired accuracy and minimizing communication overhead between simulation entities, the overall performance of simulation becomes improved.

Model partitioning is a major issues effecting simulation performance. Simulator performance can be significantly improved by optimally distributing simulation models into simulation entities before initiating the simulation process. Optimal model distribution is closely related to how models are partitioned and deployed to those

entities. Thus, it is important to develop partitioning algorithms that can find optimal or, at least, acceptable partitioning results for given simulation models.

The main objective of this research is to design and implement a class of generic model partitioning algorithms of hierarchical, modular Discrete Event Specification System (DEVS) models for distributed simulation. To attain this goal, a new hierarchical model partitioning algorithm is designed based on a cost analysis methodology. The cost analysis approach leads to algorithms that are concise, generic, and adaptable. The heterogeneous nature of models is captured and manipulated by the homogenous measure *cost*. Furthermore, the proposed algorithm minimizes model decomposition during the partitioning process and guarantees incremental Quality of Partitioning (QoP). The incremental QoP property is used to produce better partitioning results until a desired partitioning solution is attained. Partitioning improvement occurs during the partitioning process. Improvements are denoted in the partitioning tree and are easily tractable through the tree hierarchy. A class of advanced algorithms is derived from the proposed algorithm to show its extendibility and adaptability for various distributed system configurations.

1.2 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 briefly introduces background information necessary for the remaining chapters. It covers model partitioning, modeling and simulation based on the DEVS paradigm, distributed simulation, and algorithm design and analysis.

Chapter 3 describes the cost analysis methodology that plays a key role in the proposed partitioning algorithms. Major issues regarding those algorithms are presented from the perspective of cost analysis. A series of cost measures for cost generation, cost evaluation, and cost aggregation are explained.

Chapter 4 presents a new hierarchical model partitioning algorithm along with its main features. Those features include incremental quality of partitioning improvement and minimum model decomposition. Theoretical analysis and optimality issues of the algorithm are also described.

Chapter 5 lists a class of advanced algorithms derived from the proposed algorithm to show extendibility and adaptability for various distributed system configurations. Algorithms for homogeneous and heterogeneous systems based on static and dynamic information are discussed.

Chapter 6 presents applications of the proposed algorithms. As an example, a new DEVS M&S framework based on proposed algorithms is presented. Applications in the field of distributed computing and scientific computation are presented to show usability and versatility of the proposed algorithms.

Chapter 7 compares the proposed algorithms with other partitioning algorithms of DEVS models. Chapter 8 presents various experimental results regarding the proposed algorithms. Finally, Chapter 9 summarizes and concludes this dissertation.

CHAPTER 2. BACKGROUND

2.1 Model Partitioning

Model partitioning refers to the process of creating a set of partition blocks from given model(s) based on certain decision-making criteria. Depending on the structural relationship between models, various partitioning schemes can be applied. For example, if models are composed of only non-hierarchical models, a (non-hierarchical) clustering technique could be used to aggregate the models into a set of partition blocks. Instead, if a hierarchical model is to being partitioned into several partition blocks, a partitioning technique decomposing the model into a set of its component models is required.

Numerous model partitioning algorithms have been developed in the last few decades. Among them, algorithms based on graph partitioning are mainly introduced here [1, 2]. In those algorithms, a graph is applied to capture structural relationships between models, and a tree is considered as a special kind of a graph that forces unidirectional causality between two nodes and prohibits circular connections between them. In the graph, a node and a link represent a model and a relationship between associated two models, respectively.

Partitioning algorithms are mainly classified into *random partitioning algorithm*, *partitioning improvement algorithm,* and *heuristic algorithm*. The random partitioning algorithm is the simplest partitioning approach that creates a set of partition blocks with randomly selected models. The algorithm takes an insignificant amount of time to complete the partitioning task and is easy to implement. But, it may produce low quality

results. Quality of partitioning result can be improved by including domain knowledge in the algorithm [3, 4].

The partitioning improvement algorithm refers to an algorithm in which the quality of partitioning is improved as partitioning proceeds. The Kernighan-Lin algorithm performs partitioning by randomly assigning models to partition blocks at the initial time and swapping models between partition blocks during the partitioning process if swapping produces a better partitioning result[5]. Numerous partitioning algorithms derived from this algorithm are widely used in various application domains. The heuristic partitioning algorithm refers to any algorithm that uses domain-specific knowledge for performing the partitioning process.

Partitioning algorithms can be realized by utilizing certain optimization techniques. The most widely used technique is simulated annealing. Simulated annealing is a general-purpose optimization technique based on a statistical methodology[6]. Similar to the KL algorithm, models are randomly assigned to partition blocks at the initial time. Thereafter, models are swapped between partition blocks only if swapping reduces a cost function. The cost function is built by considering information on models and partition blocks. Partitioning algorithms based on evolutionary algorithms are another good example in this category.

Partitioning algorithms can also be constructed based on geometric information of models. Recursive Coordinate Bisection (RCB) and Recursive Graph Bisection (RGB) are some of them[7, 8]. In the RCB algorithm, partitioning is performed based on spatial information about models. The N-body problem is a good example of this algorithm. In

the RGB algorithm, partitioning is performed based on connectivity between models rather than spatial information. Specifically, if model topology is represented by a graph, the total number of links between two nodes (or models) is considered instead of the Euclidean distance between them.

2.2 Hierarchical Model Partitioning

Hierarchical model partitioning is the process of building a set of partition blocks by decomposing or constructing hierarchical structures based on certain decision-making criteria. The hierarchical structure is generally represented by a tree structure. In the structure, a node having no children is an atomic (or non-decomposable) node. A node having children is a coupled (or decomposable) node. The coupled node could contain other coupled nodes and a collection of atomic nodes. During partitioning, the hierarchical model structure may be dynamically permutated over time and space, if necessary.

A partitioning policy specifies what kind of partitioning approach or technique is applied to models process. There are three partitioning policies which are widely accepted and applied in many application domain problems; *flattening, deepening,* and *heuristic. Flattening* is a structural decomposition technique that transforms a hierarchical structure into non-hierarchical structure. Depending on the degree of structural decomposition, flattening is divided into *full flattening* and *partial flattening*. In *full flattening*, a hierarchical structure is decomposed until only a non-hierarchical structure remains. In *partial flattening*, both hierarchal and non-hierarchical structures coexist.

Deepening, also known as *hierarchical clustering*, is a structural aggregation technique that transforms non-hierarchical structure into hierarchical structure. It could be considered the inverse transformation of flattening because hierarchical structure is built from non-hierarchical structure. Various deepening algorithms are surveyed and evaluated in the [9]. A *heuristic* policy is an ad-hoc policy that uses techniques other than those that are associated with flattening and deepening.

2.3 DEVS Model and Simulation

2.3.1 Fundamentals of Computer Simulation

Computer *simulation* is an activity of representing temporal behavior of a physical or a conceptual system for a specific period of time. A Simulation *model* is a specification representing the system in terms of a set of states, events, and behavior functions. Simulation time is not identical to physical time. Simulation *time* is a virtual time that can run faster than physical time, slower than physical time, or equal to physical time. Also, time resolution can be arbitrarily defined.

During simulation, the current status of a model is represented by a state. A state transition occurs just before initiating or after completing a particular behavior. A state feasibility test may be involved before a state transition happens. An event is a data object that is produced and consumed by simulation components (e.g., logical simulator and coordinator). If necessary, a set of events is exchanged between those components to complete a simulation task. A behavior function is invoked when events are received or produced by a model or a specific behavior of the model is to be performed.

Simulation is classified into continuous simulation and discrete simulation depending on state transition occurrence interval. During simulation, if state transition occurs continuously in time, the simulation is a continuous simulation. Instead, if state transitions do not happen continuously, the simulation is called a discrete simulation. In discrete simulation, if state transitions occur based on discrete time intervals (or time steps), the simulation is called a time driven discrete simulation (or discrete-time driven simulation). If state transitions happen based on event activities, the simulation is referred to as an event-driven discrete simulation (or discrete-event driven simulation).

Depending on the simulation time synchronization scheme, simulation is viewed as conservative or optimistic. In the conservative scheme, time should be globally synchronized and all simulation activities at a specific time are completed before advancing time[10, 11]. In the optimistic scheme, time does not need to be synchronized globally and simulation activities at a particular time need not all be completed before advancing time, either. Only when time causality problem occurs, time needs to be synchronized[12, 13]. Conservative schemes guarantee all activities are performed without any time causality problems. By loosening the time causality constraint, optimistic schemes perform better for certain simulation problems that contain a high degree of parallelism between simulation models. However, it requires additional memory to keep information regarding activities that occurred at previous times. When time causality problems happen, current simulation time rolls back to a previous time that did not violate time causality. Generally, the performance of the simulation is not directly

associated with simulation time synchronization scheme. Rather, it is related to the nature of the given simulation problem.

2.3.2 Discrete Event System Specification

The Discrete Event System Specification (DEVS) is a formalism providing a mean of specifying a mathematical object called a system[14]. Discrete event systems represent certain constellations of such parameters just as continuous systems do. For example, the inputs in discrete event systems occur at arbitrarily spaced moments, while those in continuous systems are piecewise continuous functions of time. The insight provided by the DEVS formalism is in the simple way that it characterizes how discrete event simulation languages specify discrete event system parameters. Having this abstraction, it is possible to design new simulation languages with sound semantics that easier are to understand. For example, the DEVJAVA environment[15, 16] is an implementation of the DEVS formalism in Java that enables the modeler to specify models directly in its terms. The DEVS formalism has been applied to a number of continuous as well as discrete phenomena[17-19]. The use of discrete events, rather than time steps, as a basis for simulation has been shown to reduce computation time by orders of magnitude in many applications [18, 20, 21].

The DEVS modeling approach captures a system's structure from both functional and physical points of view. A system is described by a set of input/output events and internal states along with behavior functions regarding event consumption/production and internal state transitions. Models are classified as either *atomic* or *coupled.* The atomic model is the smallest unit for describing a system and the coupled model is an aggregation of

models. Each component of the coupled model can be either atomic or coupled. An atomic model is a non-decomposable component with well-defined interfaces. A coupled model designates how (less complex) models are coupled together and how they interact with each other.

2.3.3 Formal Description of DEVS Models

The Atomic model can be illustrated as a black box having a set of *inputs* (X) and a set of *outputs* (Y). The inputs and outputs are assumed to be events that are given to or generated by the model. The Atomic model also specifies a set of *internal states* (S) with functions (i.e., *external transition function* (δ_{ext}), *internal transition function* (δ_{int}), *output function* (λ), *and time advance function* (ta)) that describe the dynamic behavior of the model. The *external transition function* may change the state when an input event is given to the model. The *Internal transition function* computes the next state when no events have occurred. The *output function* generates an output event based on the current state. *The time advance function* adjusts simulation time after generating output events. In the DEVS formalism, the atomic model is defined by the structure

$$M = < X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta >$$

Where,

- X : a set of input events
- Y : a set of output events
- S : a set of sequential states
- δ_{ext} : $Q \times X \rightarrow S$; external transition function
  - where, $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$; set of total states
  - e is the time elapsed since last transition
- δ_{int} : $S \rightarrow S$; internal transition function
- λ : $S \rightarrow Y$; output function
- ta : $S \rightarrow R^{+}_{0,\infty}$; time advance function

Similarly, the Coupled model can be considered as a black box having a set of *inputs* and a set of *outputs*. The coupled model, unlike the atomic model, performs different tasks - routing an event that is generated from its component models to a set of other components according to coupling ($Z_{i,j}$) information. The coupling is divided into three types; *external input coupling, external output coupling*, and *internal coupling*. The *external input coupling* specifies couplings between the input of the coupled model itself and the inputs of components. The *external output coupling* specifies couplings between outputs of components and output of the coupled model. The *internal coupling* specifies couplings between outputs of components and inputs of components. Each component (M_i) of the coupled model can be either an atomic model or another coupled model. The coupled model is defined in the DEVS formalism as follows

$$DN = < X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select >$$

Where

X_{self} : a set of input events
Y_{self} : set of output events
D : set of components
M_i : a component
For each i in D $\cup$ {*self*}, i is not in I_i,
$M_i = < X_i, Y_i, S_i, \delta_i, \lambda_i, ta_i >$
I_i : the influences of i
$Z_{i,j}$: i-to-j output translation function
$Z_{slef,,j} : X_{self} \rightarrow X_j$, External input coupling function
$Zi_{,slef} : Y_j \rightarrow Y_{self}$, External output coupling function
$Zi_{,j} : Y_j \rightarrow X_j$, Internal coupling function
select : tie-breaking function

2.3.4 DEVS M&S Frameworks over Distributed Network Infrastructure

DEVS M&S frameworks have been implemented for numerous distributed network infrastructures[22]. DEVS-CORBA and DEVS-HLA are two examples[23, 24]. DEVS-

CORBA is a DEVS M&S framework that runs on industry-standard distributed network middleware, the Common Object Request Broker Architecture (CORBA)[25, 26]. DEVS-HLA is a DEVS M&S framework that is targeted for the High Level Architecture (HLA)[27, 28]. Both DEVS-CORBA and DEVS-HLA allow users to perform modeling and simulation without any knowledge of the underlaying network middleware (i.e., CORBA and RTI). A new DEVS M&S Framework over advanced network systems is addressed in CHAPTER 6.1[29].

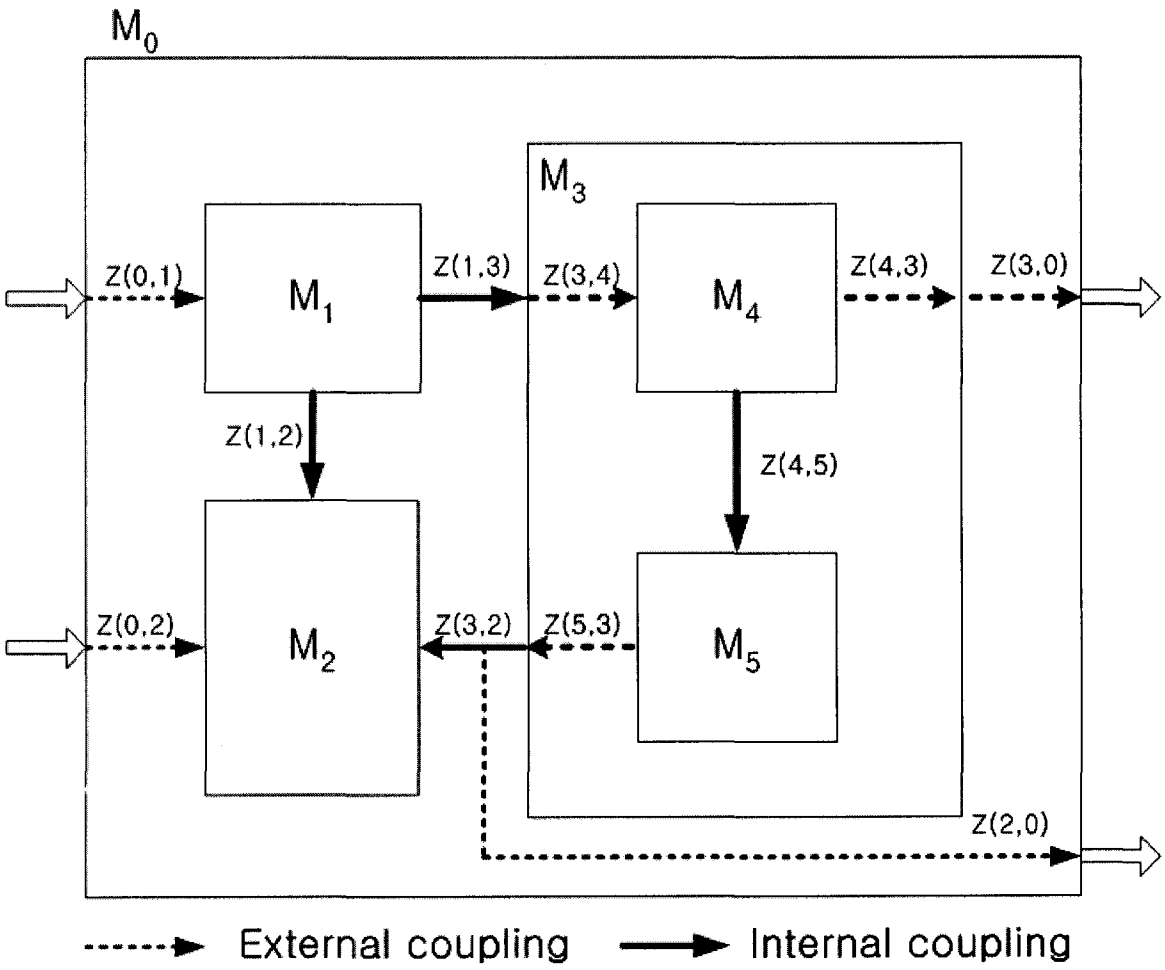


Figure 1. An Example of a DEVS Coupled Model

2.4 Distributed Network Infrastructure

Distributed network infrastructure is divided into three models; *Client-Server, Peer-to-Peer,* and *Server-Network.* Each model is briefly introduced in following sub chapters.

2.4.1 Client-Server Model

The client-server model consists of clients and a server, where the server provides services that are requested by its clients[30]. In a client/server network, a client sends a request to the server that in turn sends a reply back to the client. Interactions are many-to-one from clients to a server and one-to-many from a server to clients. In this model, clients do not communicate directly. Instead, they use the server as an intermediary. Such systems are simpler to implement when ownership disputes (e.g., which client should be given preference to update a file) arise since these can be easily handled by the server. For example, in a multi-user database, such disputes are typically resolved exclusively by the server.

Several styles of client-server architectures are used in network-enabled applications. With the emergence of the World Wide Web (WWW,) the thin-client has gained much popularity for ease of deployment and version control. Thin-clients are typically very small, meaning that they can be downloaded quickly from a network server and used in a "one shot" fashion. Versioning of the thin-client is almost trivial. When the software is recompiled, a user has automatic access to the new version when he (or she) connects to the server. The thin-client paradigm is employed by many X-Windows based applications (and the like) and, to a lesser extent, by Java applets. The thin-client, however, is not always the most suitable solution. It has several disadvantages. Chief among these is the

demand for significant bandwidth for all but small applications. Thin-clients rely on the server to do most, if not all, of the application's processing. The client's machine is mostly used to display results and to make new requests. In a highly interactive system requiring a modest amount of data transmission, this can result in frustration when the network is slow or unreliable.

The alternative to thin-client is thick-client. However, thick-clients are, arguably, more of a problem to the user (and the designer) compared with the thin-client. Nevertheless, they also tend to be more robust, scaleable, and capable of handling even the worst network conditions/constraints. The thick-client application requires that the client process maintain a local copy of the relevant data and program. This allows each client to act more or less independently. depending on how empowered the thick-client is. A server provides services that the client cannot provide by itself. This normally includes resolving ownership disputes and acting as a "switch-board" for clients that want to communicate across the network. Thick-clients also have the advantage of being scaleable. By requiring each client to do its own processing, the server is free to perform other, usually communication related, tasks that are necessary to keep a large user base productive and happy.

However, there is a cost for thick-clients since client-side applications are large and complex. For the development team, there are new issues concerning synchronization across a network (as opposed to thread synchronization in a single address space). In a system with mobile objects, object naming becomes more complicated as well.

For the user, the larger program size can mean lengthy download times, and tends to discourage users from updating to newer versions of the software when it becomes available. However, a thick-client can make the network invisible to the user while the software is being used. Therefore, the type of application usually determines whether thin-client or thick-client is the most suitable solution given the available hardware/software technologies.

2.4.2 Peer-to-Peer Network Model

Despite widespread use of client-server network environments, they are not suitable for all applications. In large-scale systems, the server can become a communications bottleneck, as all network traffic has to pass through a single processing node. In such a setting, the server acts as a narrow bridge during a traffic jam. It might take five minutes to drive to the bridge, 3 hours to cross it, and then another 2 minutes to reach the final destination. Another problem with the client-server configuration is the introduction of a single point of failure. If the server crashes, the entire system shuts down.

To resolve these problems, *Peer-to-Peer* (serverless) network models have been introduced[31]. In such a model, all clients (or subsets thereof) are directly interconnected to one another. In such an arrangement, there is no intermediary and the client broadcasts (or multicasts) its messages directly to every client in the network. This removes the single point of failure and eliminates the server bottleneck. However, it introduces ambiguities when any dispute arises. When a decision must be made, some process (owned by one or more clients) must take responsibility for its resolution. Several election algorithms have been proposed to resolve such ambiguities. The peer-to-peer

network has advantages in terms of performance, but the price is communication complexity and synchronization. To deploy a peer-to-peer collaborative architecture, some kind of middleware (e.g., CORBA) is essential to overcome the fundamental communication and synchronization problems.

2.4.3 Server-Network Model

The server-network model provides a hybrid architecture based on both the client-server and the peer-to-peer model. Groups of clients are connected to a server. Several servers are interconnected. A client in server-network may access local servers as well as remote servers. This is a distinctive feature of the server-network. It also provides improved scalability over the client-server model and better control than the peer-to-peer model. Once again, the enhanced features of a server-network must be traded off against additional complexity that is necessary for its maintenance.

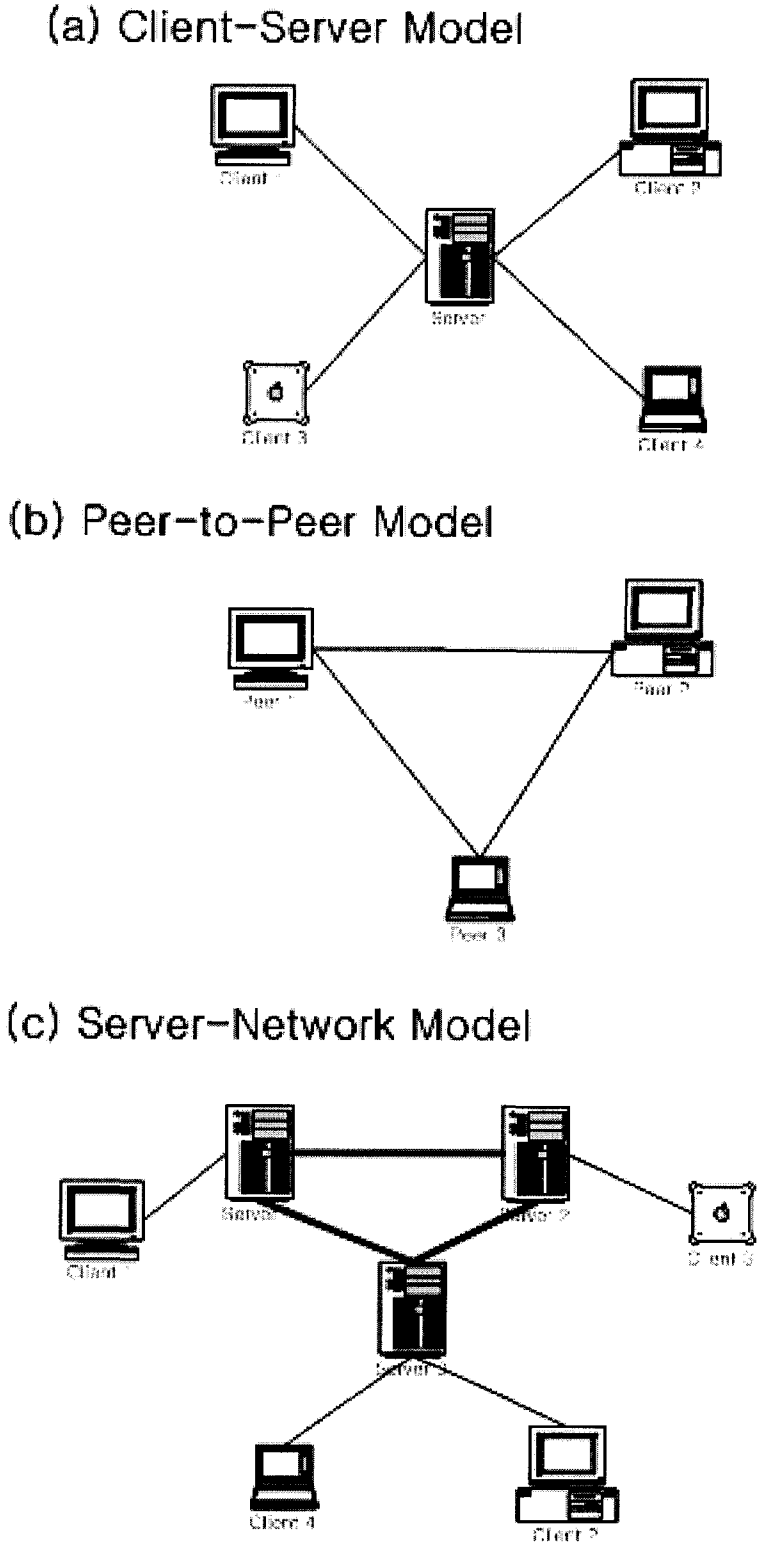


Figure 2. Distributed Network Models; client-server, P2P, and Server-network

2.5 Algorithm Analysis and Design Techniques

2.5.1 Approaches to Algorithm Analysis

Algorithm analysis is used to estimate or understand of the performance of a given algorithm. Numerous algorithm analysis techniques exist[32-36]. These techniques are mainly divided into two types; complexity analysis and execution time analysis.

The (computational) complexity analysis is an analytical approach of determining the asymptotic *growth rate* (or the order of growth) of an algorithm particularly in the worst case [33]. In this approach, an algorithm is presented by its upper bound (i.e., *O)*, lower bound (i.e., Ω), and the growth rate (i.e., Θ). Nevertheless, the growth rate is mainly of interested. The notations, *O*, Ω, and Θ, are formally defined as follows[37]

Definition 1:

Given a function *f(N),*

$O(f(N))$: a set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from *above* as $N \rightarrow \infty$

$\Omega(f(N))$: a set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from *below* by a (strictly) positive number as $N \rightarrow \infty$

$\Theta(f(N))$: a set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from both *above* and *below* as $N \rightarrow \infty$

The execution time complexity analysis is an analytical approach of determining the performance of an algorithm in terms of its best case, worst case, and average case execution time[38, 39]. Compared to the complexity analysis, the approach allows obtaining very precise results. However, it may require sophisticated analytical techniques.

2.5.2 Algorithm Design Patterns and Techniques

An algorithm can be designed based on a certain design pattern[40]. It is convenient to use a particular design pattern to implement a desired algorithm instead of creating the algorithm from scratch, if an appropriate pattern exists. It reduces considerably the amount of time required to build and test the algorithm. Even if a pattern does not exit, the algorithm could be realized by applying a particular algorithm design technique.

Recursion is a powerful design technique. By invoking same algorithm with different data sets, redundant operations could be reduced considerably. First-order recursion and Dived-and-Conquer (DC) are some of the algorithm design techniques based on recursion. *Graphs and trees* are also widely used design techniques. By transforming a given problem into a graph or tree and applying well-known techniques such as first-cut and depth-first-search, the desired algorithm could be easily produced. *Dynamic programming* is another design technique that is mainly used to solve combinational optimization problems. It stores computational results of a partial solution into a table and reuses those results instead of generating identical results, if necessary, until algorithm terminates. By storing recurrent partial results, this technique reduces execution time. However, it requires additional memory space to store those results. *Randomized techniques* are based on non-deterministic approaches. Unlike deterministic algorithms, it selects a partial solution randomly and proceeds until a certain solution is attained. Evolutionary algorithms are a good example. Randomized algorithms may reduce considerably the amount of time and space required to solve a problem. However, it guarantees neither deterministic execution time nor deterministic results.

CHAPTER 3. COST ANALYSIS METHODOLOGY

Cost analysis is an analytical approach of abstracting heterogeneous *resource* information into homogeneous *cost* information and performing certain analytical operations with respect to the cost information. The methodology covers a wide spectrum of analytical activities ranging from harvesting cost information to performing sophisticated analytical operations regarding the information. Specifically, it copes with how to acquire cost information from particular resource information *(cost harvesting)*; how to create cost information based on a certain generation scheme *(cost generation);* how to coalesce a group of small costs into a bigger cost *(cost aggregation)*; how to compare a cost with another cost *(cost evaluation)*; how to analyze cost information *(cost analysis)*, etc. In this chapter, various cost measures and major issues regarding the above analytical activities are briefly addressed. The methodology plays a key role in a class of partitioning algorithms proposed in chapter 4 and 5.

3.1 Cost Measure

A cost measure is a conceptual metric that captures heterogeneous resource information in terms of cost. Various metrics could be used to harvest, generate, and evaluate cost information based on distinct decision-making criteria. *Complexity*, *I/O connectivity*, and *behavior functions* are some of the cost measures that are widely used.

The *complexity* is a classical cost measure. It is based on the assumption that cost is highly correlated with complexity. For example, the cost of a model increases as model complexity grows. Model complexity could be estimated by counting the total number of

internal states of the model or measuring the activity of the model for a certain period of time. The number of internal states is a *structural* complexity measure. Model activity is a *computational* complexity measure.

The *I/O connectivity* might be a better alternative if cost information is mainly associated with I/O activities. For example, suppose a system that spends most of its computational power to handle incoming service requests and the service time for every request is identical (such as WEB authentication server). Performance of the system is generally judged by I/O activity rather than system complexity. An example of I/O connectivity measure is the total number of active connections.

The *behavior function* captures computational activities much better than static cost measures. It is a dynamic cost measure dealing with a behavioral property of a system that varies over time. For example, every load balancing system provides a set of behavior functions for detecting nodes under heavy load (i.e., load detection) and migrating a task from one host to another (i.e., task migration). The performance of the system is captured more accurately by monitoring the activity of those functions.

The mathematical interpretation and realization of the cost measures is presented in chapter 3.2 followed by addressing analytical activities in the cost analysis methodology.

3.2 Cost Function

A cost function is a mathematical representation of a cost measure. It is an abstract function. Thus, for a single cost function, numerous implementations are possible. It is notable that the actual implementation of a cost function is associated with decision-making criteria and application domain characteristics. For example, suppose there is a model representing a system with a set of I/O interfaces, X *and* Y, a set of internal states, Γ, and a set of behavior functions, Λ. Various cost measures and their corresponding cost functions are applied to the model based on certain decision-making criteria. If the system is a router propagating incoming messages received through its input channel into a particular output channel, the IO connectivity measure is regarded as an appropriate cost measure. The total number of the I/O interfaces (i.e., *Size(X)* + *Size(Y)*) and connectivity space of the I/O interfaces (i.e., *Size(X)* * *Size(Y)*) are some of possible cost functions regarding the cost measure. If the system is a server that uses several pipeline stages to finish its incoming service requests, service time in every stage is identical, and each stage is captured as an internal state in the model, the complexity measure associated with internal states could be used as an appropriate cost measure for the system. The total number of internal states (i.e., *Size(Γ)*) is a possible cost function for the measure. Some cost measures and their corresponding cost functions are listed in Table 1.

Table 1. An Example of Cost Measures and Their Corresponding Cost Functions

Cost measure	Cost function	Decision-making criteria
I/O connectivity	$Size(X_{model})*Size(Y_{model})$	The cost of a system is generally proportional to the number of I/O interfaces if the system is dedicated to serving I/O requests.
System Complexity (without I/O connectivity)	$Size(\Gamma_{model})$	The cost of a system is represented by the number of internal states rather than the number of I/O access points if system performance relies on its complexity.
System Complexity (with I/O connectivity)	$Size(\Gamma_{model})*$ $Size(X_{model})*Size(Y_{model})$	The cost of a system can be captured more appropriately by considering both I/O interfaces and system complexity
System Activity	$Numberof(Transition_{model})$	The cost of a system can be capture more appropriately by considering dynamic system behaviors.

3.2.1 Implicit verses Explicit

The cost of a model could be obtained *implicitly* or assigned *explicitly*. If cost information is directly retrieved from the model by invoking a particular function on the model, the function is referred to as an *implicit cost function*. Instead, if cost information is generated and assigned to the model using a function, the function is referred to as an *explicit cost function*. For example, suppose a model having a function returning the total number of internal states as its cost. Cost information of the model is easily acquired by accessing the function. The function is an implicit function. Assume a function creating a

positive integer and assigning this number to a model and the model stores it as its cost. The function is an explicit function.

The implicit function paradigm forces a model to perform its own cost evaluation when the model is involved in cost harvesting (or retrieval) because the process of cost evaluation is included in the model. It permits collecting cost information from the model without deliberating a specific cost function. The implicit function is suitable for handling a system that is a compound of numerous heterogeneous entities because any domain-specific knowledge or cost retrieval mechanism regarding those entities is not required to harvest cost information from the system. The most important task regarding the implicit function is to find appropriate cost measures (e.g., complexity, I/O connectivity, etc.) associated with models for capturing cost correctly and efficiently.

The explicit function enables characterizing models in term of cost distribution. It allows building a certain cost pattern dispersed over models and enables testing or controlling those models by customizing the function. In the perspective of model, the model only needs to contain cost information that is supplied from outside instead of computing its own cost information. Unlike the implicit function that needs only cost evaluation, the explicit function requires both cost generation and cost assignment. In the explicit function, it is vital to deliberate appropriate schemes regarding cost generation and cost assignment. Advanced mathematics and heuristics may be applied to realize those schemes.

Each cost function introduced in Table 1 becomes implicit by integrating itself into a model. It also becomes explicit by generating cost information regarding its associated

cost measure and assigning the information into the model. Figure 3 shows an example of cost generation and cost assignment in an explicit cost function.

3.2.2 Cooperative verses Non-cooperative

With respect to a model, a cost function may require additional information about adjacent models to compute its cost. A cost function is referred to *cooperative* if it requires information about other models to find the cost of a model. A cost function is referred to *non-cooperative* (or *isolated*) if it needs only one model to compute the cost of that model. A cost averaging function is a good example of cooperative cost function. For a given model, the function aggregates the costs of its neighboring models, computes the average of those costs, and assigns average to the model. Cost functions using information about the communication between models could be considered a cooperative scheme.

3.2.3 Cost Evaluation versus Cost Generation

Cost evaluation is the process of extracting cost information from a model. The cost information is acquired by accessing a particular function on the model. With an implicit function, the model executes its own cost evaluation mechanism and returns the result as its cost. With an explicit function, the model simply returns the cost assigned to it. It is suitable for handling heterogeneous models because costs of models are easily retrieved from the model without any knowledge regarding those models.

Cost generation is the process of producing cost information using a certain generation scheme. Mathematical and heuristic schemes are used to create cost

information. Generated cost information is eventually assigned to models based on a particular cost assignment scheme. It is appropriate for creating models satisfying particular cost distribution or pattern. It allows modeling and testing a system convenient from the perspective of cost analysis. For example, if a new system is required to be designed and deployed, it is convenient to create and test the system using cost generation scheme with respect to various system configurations. It is because cost patterns of various configurations are easily produced using this scheme.

3.2.4 Deterministic versus Non-Deterministic

A cost function is implemented using deterministic or non-deterministic techniques. If cost information is evaluated or generated using a deterministic technique, it is represented by a deterministic object such as a single value, a set of discrete objects, a range of continuous values, etc. Instead, if a non-deterministic technique is involved in cost evaluation or cost generation, the cost information is represented by a non-deterministic object such as a probability distribution or stochastic process. Some cost generation techniques are listed in Table 2.

Table 2. Various Cost Generation Techniques [41-44]

	Major approach	Examples
Deterministic	Iterative function	Fractal Geometry
	Differential equation	Non-linear differential equation
	Transformative function	Fourier transform
	Logarism	Natural log
Non-deterministic	Probability	Probability Density Function (PDF)
		Probability Mass Function (PMF)
		Cumulative Distribution Function (CDF)
		Cumulative form of PDF or CDF
	Stochastic process	Brownian motion
		Stable process
		Poisson process
		Levy process
	Statistical function	Random sampling
		Confidence interval
		Hypothesis testing
Heuristic	Evolutionary algorithm	Genetic algorithm
	Uncertainty	Chaos, fuzzy logic

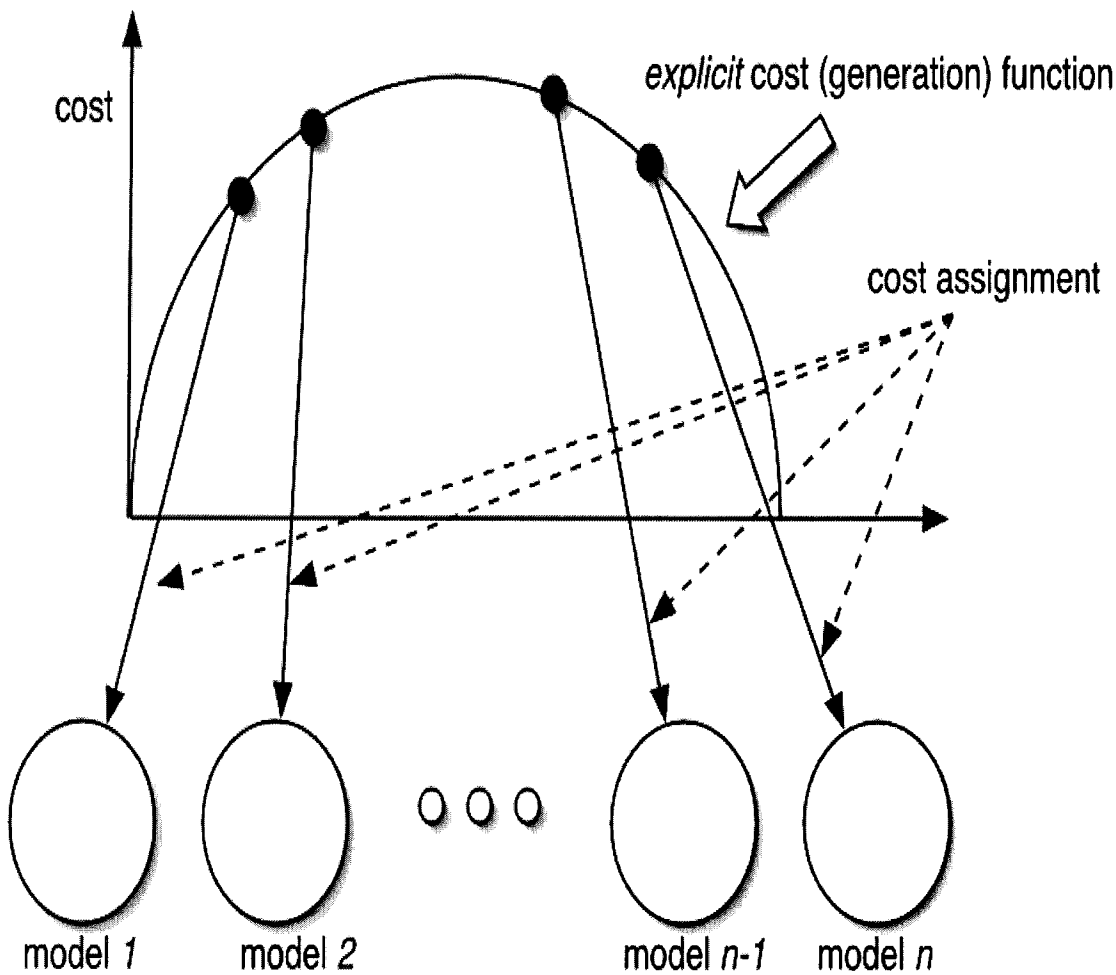


Figure 3. An Example of Cost Generation and Assignment in an Explicit Cost Function

3.3 Cost Aggregation

Cost aggregation is the process of coalescing a group of small costs into a bigger cost. By representing a group of costs by its aggregated cost, sophisticated relationships between costs can be significantly simplified. This makes manipulating cost information much easier with respect to cost analysis. Similar to cost generation, the result of cost aggregation could be represented by a single value, a set of discrete values, a range of continuous values, a probability distribution, or a stochastic process. Cost aggregation is applied to either hierarchical model or non-hierarchical model structures. Cost aggregation that is associated with hierarchical model structures is discussed in this dissertation.

Cost aggregation is associated with cost granularity and cost homogeneity. When a set of costs is aggregated into a bigger cost, cost granularity increases and cost homogeneity decreases. The cost granularity and the cost homogeneity are proportional to and reciprocal to cost disparity, respectively. Generally, the granularity increase when smaller costs are aggregated into a larger cost while the homogeneity increases when a larger cost is decomposed into a group of smaller costs.

Figure 4 shows cost aggregation results when some example aggregation schemes are applied to cost information that is represented by a hierarchical tree. Various cost aggregation techniques based on mathematical and heuristic approaches are presented in Table 3. Those approaches are described in following sub chapters with examples. Their main advantage and disadvantage are also addressed.

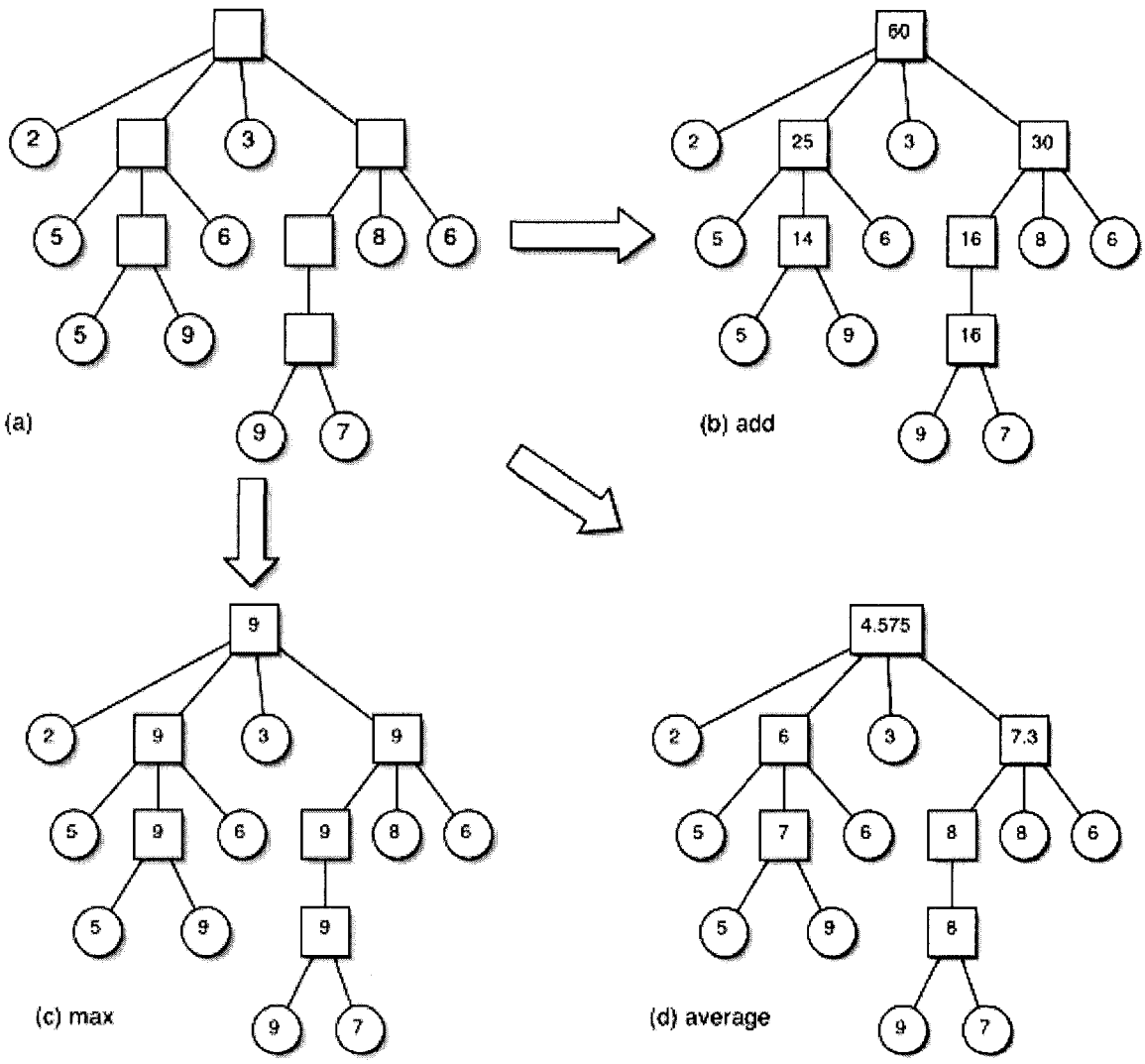


Figure 4. Effects of Various Cost Aggregation Schemes

Table 3. Various Cost Aggregation Techniques [41-43]

Major approach	Examples
Arithmetic aggregation	Addition, subtraction, multiplication
Logical aggregation	Max, min
Stochastic aggregation	random walk, entropy, estimation, ergodic,
Statistical aggregation	average, variance, maximum-likely hood,
Transformative aggregation	logarithm addition, logarithm multiplication
Geometric aggregation	
Heuristic aggregation	

Arithmetic aggregation is an aggregation scheme that coalesces a group of costs into a single cost by applying arithmetic operations. It is a widely used aggregation scheme in many application domains. Summation is the most popular technique. Summation adds all costs into a single cost. If, for example, cost aggregation is applied to a countable resources (e.g., available memory), summation is an appropriate cost aggregation method.

If cost aggregation is applied to model complexity, multiplication is superior to summation as an aggregation scheme because multiplication represents cost aggregation regarding complexity rather than summation. A major problem with multiplication-based aggregation is *cost exploding* that increases the aggregated cost very rapidly as aggregation proceeds. Figure 5 shows an example of the cost exploding for the given cost information. To avoid or prevent the problem, modified-multiplication or transformative schemes (e.g., logarithm summation) are used instead of a pure arithmetic multiplication scheme. The arithmetic aggregation scheme is easy to use and to manipulate. However, it may not appropriate for aggregating non-deterministic cost information.

Transformative aggregation is an aggregation scheme coalescing a group of costs into a single cost by applying a certain transformative operations. It makes describing and manipulating sophisticated cost information much easier by transforming original cost information into simplified or manageable alternative forms. If an inverse transform is available for the aggregation scheme, the aggregation result can be presented in original cost representation format using the inverse transform. The most well known transformative aggregation technique is the logarithm. The cost exploding problem in arithmetic multiplication aggregation is solved by transforming arithmetic multiplication to logarithm addition.

Probabilistic aggregation and stochastic aggregation (or integration) are aggregation schemes coalescing a group of costs into a single cost based on non-deterministic aggregation techniques. When cost information is not described by deterministic cost (e.g., single value, a set of discrete objects), coalescence of a group of costs cannot be achieved by deterministic aggregation techniques. Instead, an alternative scheme should be provided to cope with non-deterministic cost information (i.e., non-deterministic aggregation scheme).

Probabilistic aggregation and stochastic aggregation are two possible choices. Generally, the probabilistic aggregation is used for aggregating time-invariant non-deterministic costs. Stochastic aggregation is applied to non-stationary non-deterministic costs. In a probabilistic aggregation scheme, cost aggregation result is represented by a probability distribution. Similarly, in a stochastic aggregation scheme, cost aggregation result is represented by a stochastic process.

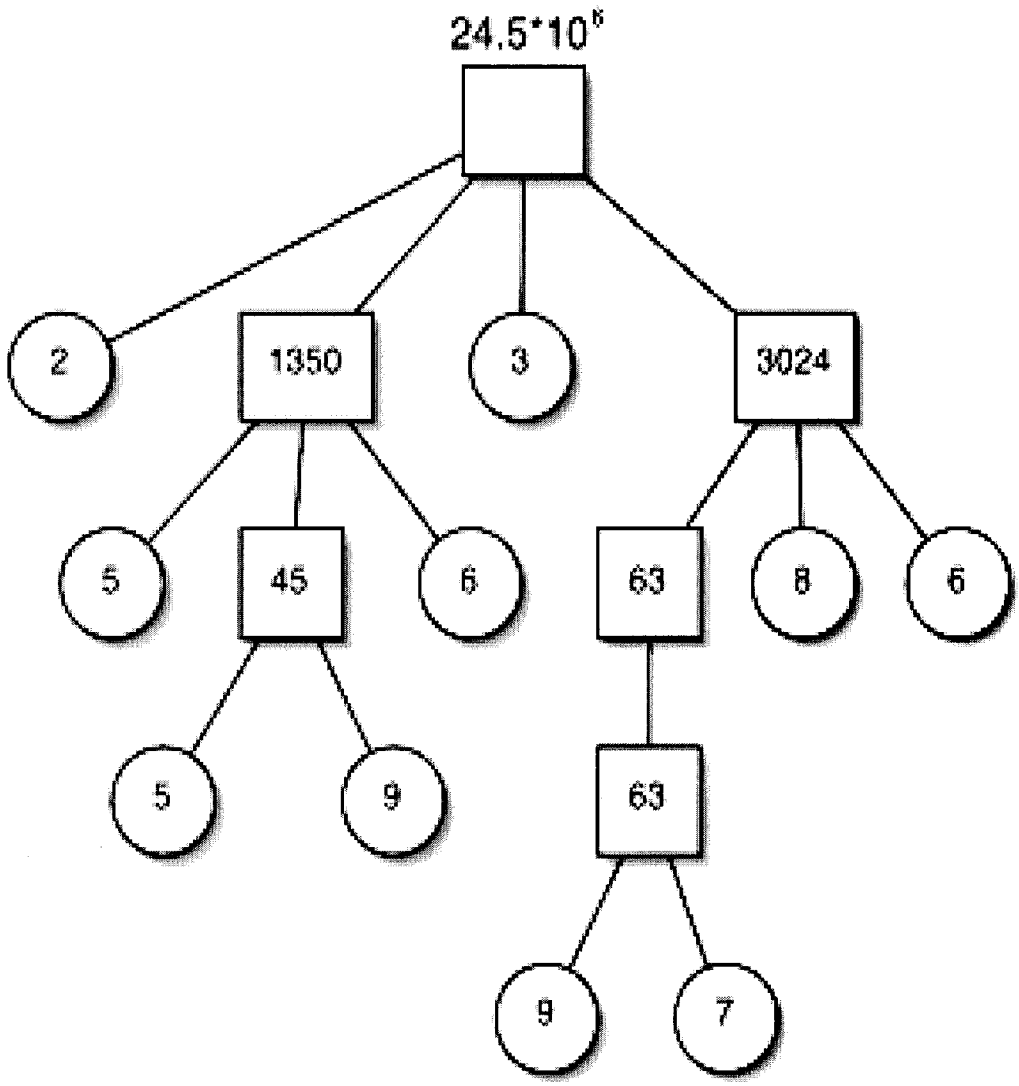


Figure 5. Cost Exploding in Arithmetic Aggregation based on Multiplication

3.4 Cost Tree

A cost tree is a tree structure that is created from a model that contains cost information. A node in the tree is either *atomic* or *coupled.* An atomic node is a terminal node containing no child nodes. A coupled node is an intermediate node holding more than one child node. Each node contains a *model* and *its cost* regardless of its classification type (i.e., atomic or coupled). The cost of the node is implicitly retrieved from its associated model or explicitly assigned to the node.

A node is evaluated by retrieving the cost information of the node. An atomic node contains its own cost only and a coupled node keeps all costs of the node and its descendents that are reachable through the tree hierarchy. Thus, the cost of a subtree starting from a particular node is acquired by simply retrieving cost information of the node without further expansion of the tree. It reduces considerably the amount of time and space required for parsing all descendants of the node and aggregating their costs during the cost evaluation process.

A cost tree is built using a recursive algorithm, Algorithm 1, applied to a given hierarchical model. Cost tree construction is initiated by creating a root node of the tree. The root node is created with a root component of a given model and its associated cost. Given the root, the algorithm recursively visits each component of the model and creates a node in the cost tree. The model of the node is linked to the component and the cost of the node is extracted from the model. The actual mechanism for cost generation and assignment with respect to the model is not described in the algorithm.

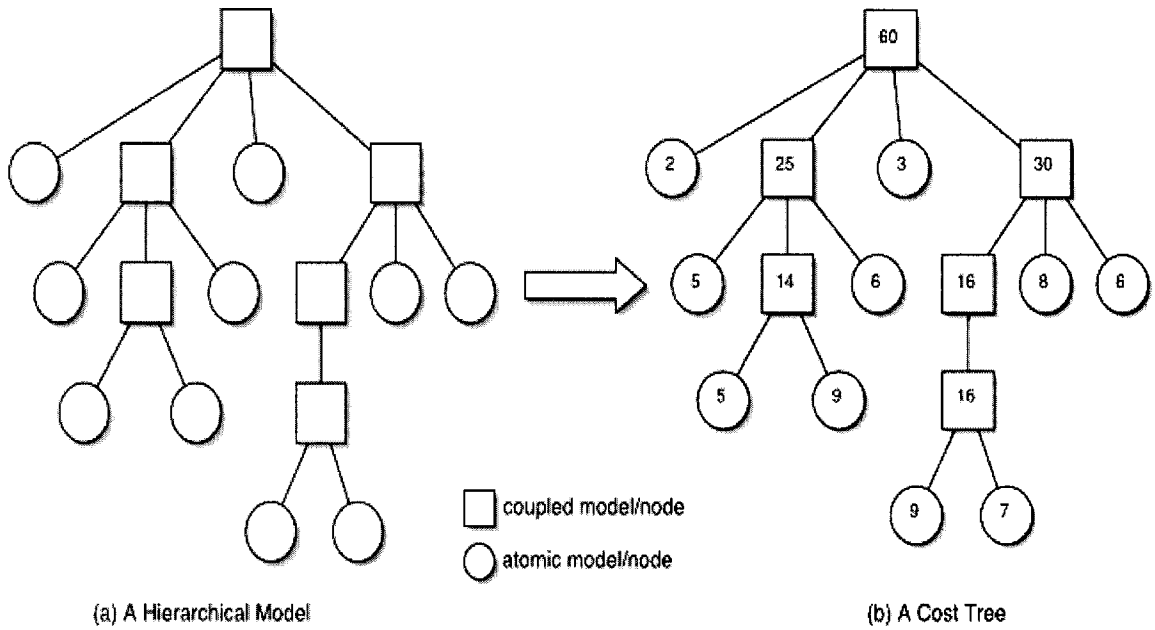


Figure 6. An Example of a *DEVS* Coupled Model and its Transformation into a Cost Tree

```
procedure cost buildCostTree(Node n)
1:    m ← get a model from the given node, n
2:    if the model, m, is coupled then
3:       c-list ← all children nodes of the model, m
4:       for each component, comp_i, in the component list, c-list, do
5:          child ← create a new node with a model of comp_i
6:          cost-of-children ←
7:                Cost-aggregation(cost-of-children, buildCostTree(child))
8:       endfor
9:       cost-of-node ← Cost-aggregation(cost-self(), cost-of-children)
10:   else // the model is atomic model
11:      cost-of-node ← cost-self()
12:   endif
13:   return cost-of-node
14: endprocedure
```

Algorithm 1. Cost Tree Construction Algorithm

CHAPTER 4. A GENERIC MODEL PARTITIONING ALGORITHM

A new Generic Model Partitioning (GMP) algorithm for a hierarchical, modular Discrete Event System Specification (DEVS) model is proposed in this chapter. The proposed algorithm decomposes a given hierarchical model into a set of partition blocks and provides solutions for distinct partitioning problems based on the cost analysis methodology. It also minimizes model decomposition during the partitioning process and guarantees *incremental* quality of partitioning (QoP) improvement until a best partitioning result is attained.

The algorithm is characterized by a set of generic cost measures for cost generation, cost evaluation, and cost aggregation. Since a cost measure is a parametric method, subject to certain axioms, the algorithm is generic and applicable to any family of models provided there is a way to manipulate the appropriate cost information. *Activity* is one of possible cost measure. However, the more general concept potentially includes other important determiners of simulation work such as number of messages sent and received. By applying one or more cost measures, a model is abstracted to a cost regardless of its complexity or heterogeneity.

The homogeneity of the cost allows the proposed algorithm to be applicable to heterogeneous problems by simply changing cost measures without any modification of the algorithm itself. This is due to the homogeneous nature of the cost analysis methodology. Thus, the proposed algorithm is highly adaptable and can be applied to various application domains.

With minimization of model decomposition, the suggested algorithm becomes less sensitive to the depth or the complexity of a hierarchical model. It makes the algorithm much more flexible and scalable compared to other partitioning algorithms based on full decomposition (or flattening). The suggested algorithm reduces considerably the number of model decompositions by preserving the original hierarchical structure of the model as much as possible. Generally, the original hierarchical structure of the model is not conserved in hierarchical model partitioning. This is because the model is decomposed and mapped into a set of partition blocks. The easiest way to achieve this goal is to flatten the model until a non-hierarchical structure exists and to perform (hierarchical) clustering on the flattened models until a reasonable partitioning result is found. The proposed algorithm minimizes model decomposition by breaking the given model only until a best partitioning result is attained, instead of fully decomposing the model before or during the partitioning process. In the proposed algorithm, a newly obtained partitioning result is compared to a previous result at the end of the partitioning process. If the new result is superior to the previous one, model decomposition is performed again to see if a better result is obtained. Otherwise, the previous result is identified as the best partitioning result. A part of original hierarchical structure could be preserved in the partitioning result if the given model is not fully decomposed when the result is attained.

One unique feature of the proposed algorithm is its support for incremental QoP improvement that guarantees partitioning results evolve into the best result without any degradation of QoP during the partitioning process. This allows the algorithm to produce

a high degree of QoP for the given model. The QoP is easily traceable through a partitioning tree hierarchy.

4.1 Partitioning Tree

A partitioning tree is a tree structure that is created and updated during the partitioning process. It contains partitioning results produced by the partitioning algorithm during the partitioning process. It shows both causality between results and incremental QoP improvement through the tree hierarchy. In the tree, a child node always represents a more recent and better partitioning result as compared to its parent node. Causality and QoP between sibling nodes are not specified.

Definition 2:

> Φ_p: A tree structure created and manipulated during the partitioning process. The partitioning tree, Φ_p, is created with a cost tree, Φ_c, and the number of partition blocks, b. A partitioning node, η, in the tree contains a collection of b partition blocks, $\{B_1, B_2, \ldots, B_b\}$, and a cost evaluation value, φ, associated with those partitions blocks. That is, $\eta = \{\varphi, \{B_1, B_2, \ldots, B_b\}\}$.

Initially, the partitioning tree is constructed by creating a root node of the tree based on a cost tree and the requested number of partition blocks. The cost tree is built from a given hierarchical model as described in the previous chapter and the number of partition blocks is provided by a user. From the root node, child nodes are created, evaluated, and expanded until a best partitioning result is achieved. A partitioning node consists of a set of partition blocks and its evaluation value. A partition block is comprised of a collection of cost tree nodes and a cost aggregation value associated with the collection.

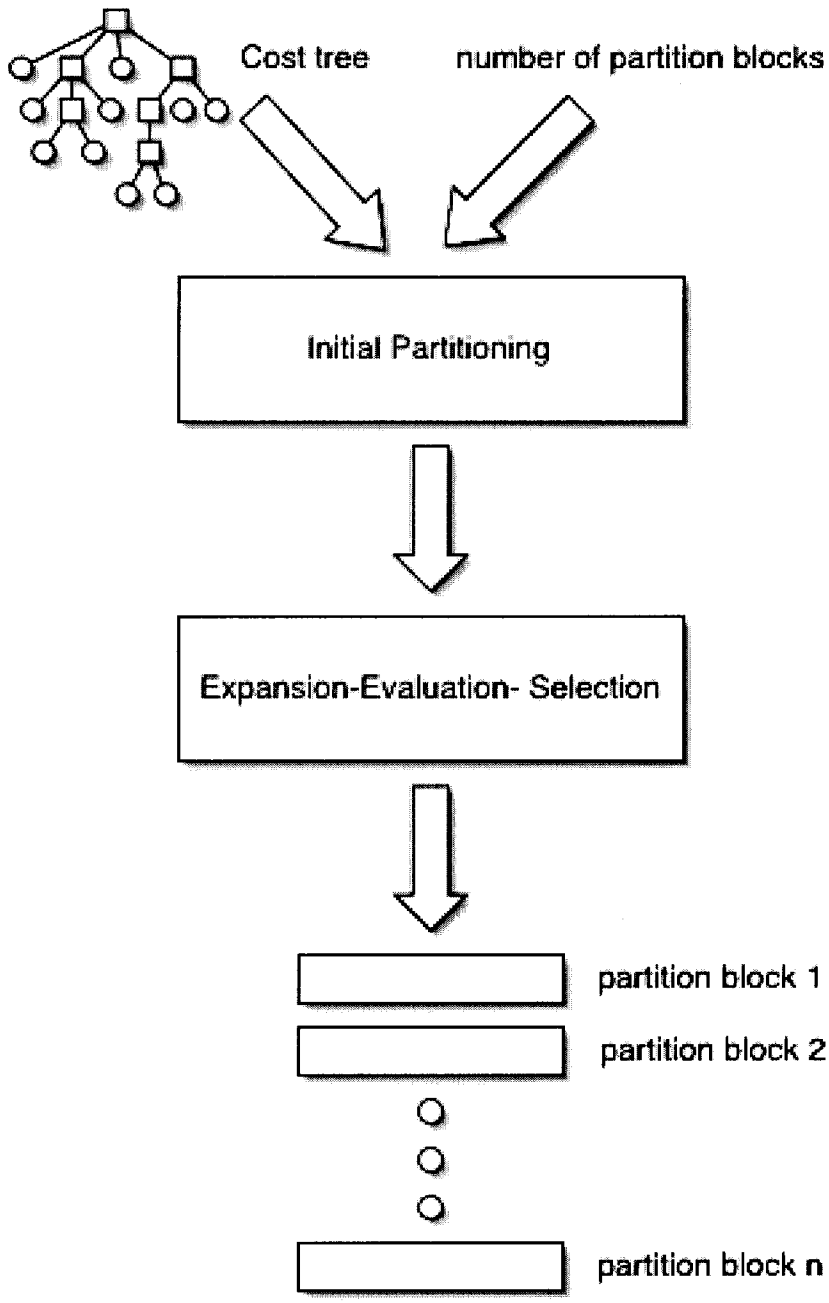


Figure 7. Life Cycle of a Generic Model Partitioning Process

The partitioning node is evaluated by a cost measure, the *partitioning evaluation function*. This function represents QoP in terms of cost by assessing partition blocks of partitioning node based on a certain cost evaluation technique. *Disparity* is one of possible cost measures aimed at evaluating the partitioning node. A partitioning evaluation function based on disparity computes cost differences between a max partition block and a min partition block of a partitioning node. The max partition block and the min partition block contain the highest cost and the lowest cost, respectively. Those blocks are easily identified by sorting all partition blocks in an order based on their costs. The partitioning evaluating function, like a cost function, is a highly flexible and generic cost measure. Various heuristic measures could also be used to evaluate or capture cost information of a partitioning node.

The incremental QoP improvement is a unique feature of the algorithm. The algorithm always produces improved partitioning results from a previous one as the algorithm proceeds. If no enhanced partitioning result exists, the algorithm terminates. Otherwise, partitioning is continued until a better alternative is achieved. Only improved partitioning results survive and are stored in partitioning nodes during the partitioning process. It is also guaranteed that the partitioning result of a child node is always superior to the result of its parent. Thus, the partitioning improvement is easily traceable through the tree hierarchy. It is notable that the partitioning tree stores partitioning results and shows how QoP improvement occurred during partitioning process, while the cost tree captures cost information of a hierarchical model.

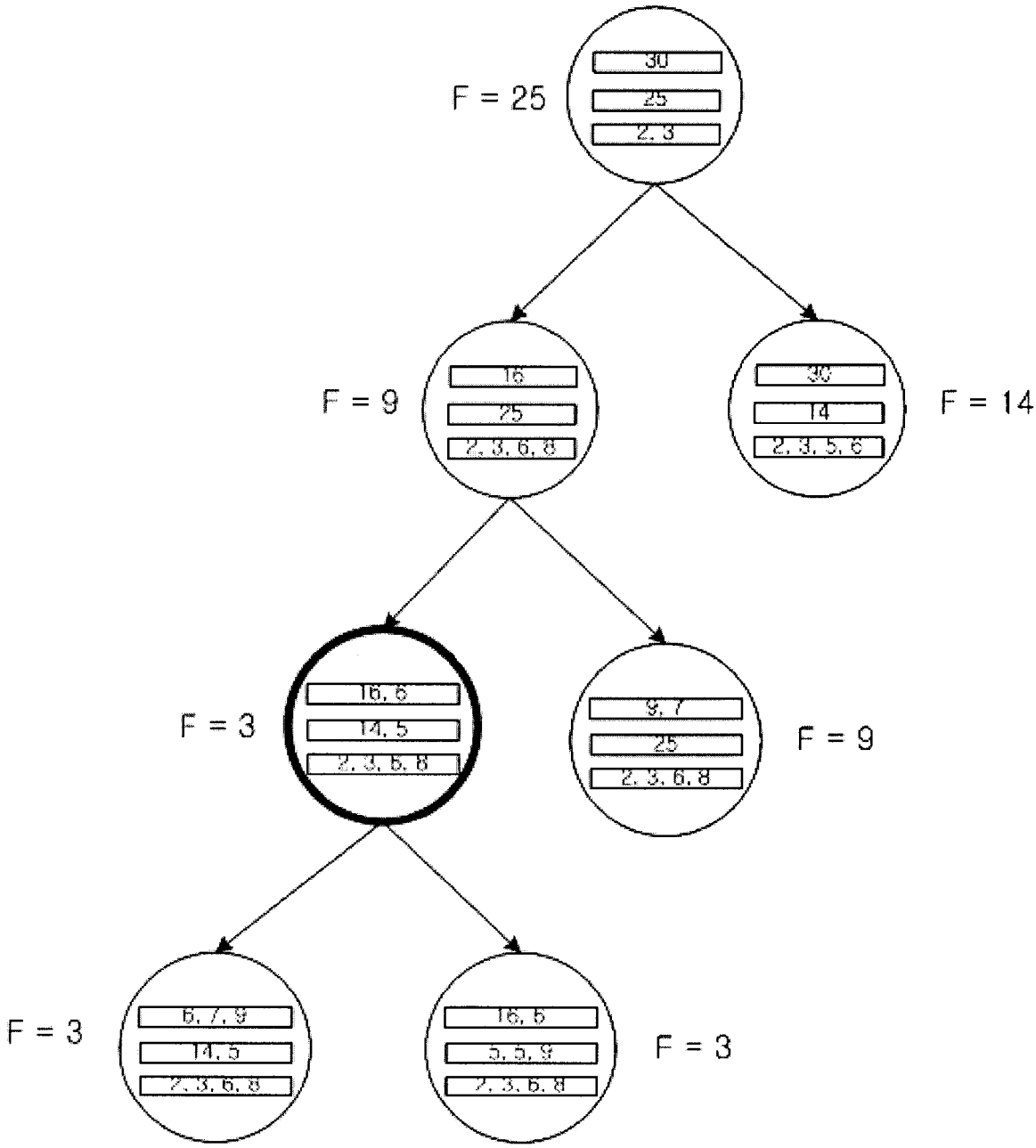


Figure 8. An Example of a Partitioning Tree When the Number of Partition Blocks is 3

4.2 A Generic Model Partitioning Algorithm

A generic model partitioning algorithm is a model partitioning algorithm that decomposes a hierarchical model into a set of partition blocks based on cost information for the model. It has two phases; *initial partitioning* and *evaluation-expansion-selection (E^2S) partitioning*. In the initial partitioning phase, a root node of the partitioning tree is constructed. While, in the phase of E^2S partitioning, child nodes of the tree are constructed, evaluated, and expanded until a best partitioning result is attained. This chapter describes those phases in depth, with some examples.

4.2.1 Initial Partitioning

Initial partitioning creates a root node of the partitioning tree. The root node is built based on a cost tree and the number of partition blocks as described in the previous chapter. The initial partitioning is started by creating a set of empty partition blocks. Once the partitioning blocks are created, each block will be populated with one or more cost nodes. Those nodes are obtained from the cost tree by decomposing a specific node of the tree. The total number of cost nodes increases or decreases during the partitioning process. Specifically, before assigning a cost node to an empty partition block, the total number of available cost nodes is compared to the requested number of partition blocks. If the number of nodes is smaller than the number of partition blocks, node expansion occurs. The node having the highest cost among the available coupled nodes is selected and expanded. Nodes expanded from the selected node become available along with existing cost nodes. Expansion is repeated until the total number of cost nodes is equal to

or larger than the number of partition blocks. After expansion, every partition block is filled with a cost node. Once all partition blocks become non-empty, the remaining cost nodes are distributed into those blocks based on some decision-making criteria. Evaluation of the initial partitioning result is done by applying a partitioning evaluation function to those blocks. The initial partitioning algorithm is presented in **Algorithm 2.**

The initial partitioning algorithm is divided into four phases; *initialization*, *expansion*, *filling*, and *distribution*. In the first phase, *initialization*, two data objects, a component list, *c-list,* and a collection of partition blocks, *p-array*, are created and initialized. The *c-list* is a bag containing cost nodes that will be obtained from the given cost tree during the partitioning process. The bag grows when a cost node of the bag is expanded and shrinks when a cost node of the bag is assigned to a partition block. The algorithm terminates when the bag becomes empty. The *p-array* is an array of bags in which each bag represents a partition block. In this phase, the *c-list* is populated with child nodes of the root node of the cost tree (Line 2). While, the *p-array* is initialized to empty (Line 3).

In the second phase, *expansion*, cost nodes of the *c-list* are removed and expanded until the length of the *c-list* is equal to or larger than the size of the *p-array*. When a node is expanded, it is removed from the *c-list* and its children nodes are stored back to the *c-list*. Specifically, the length of *c-list* is compared to the size of *p-array* (Line 5). If the former is larger than the latter, existence of a coupled node in the *c-list* is checked (Line 6). If the coupled node is found, a coupled node having the highest cost is selected and removed from the *c-list* (Line 7). The removed node is expanded and its children nodes are stored back to the *c-list* (Line 8). If no coupled node exists in the *c-list*, the algorithm

is terminated with an error message (Line 10). It is repeated until the length of *c-list* is not less than the size of *p-array* (Line 5 – Line 12).

In the third phase, *filling*, every empty partition block is populated with a cost node. The node having the highest cost is selected and removed from the *c-list* and assigned to any empty partition block. When identifying the node, only the cost value is considered. A partition block is filled with either a coupled or an atomic cost node. This property allows maintaining nodes in a homogeneous way based on the cost value regardless of the classification type of the nodes. In the algorithm, a node having the highest cost is removed from the *c-list* (Line 15) and is assigned to any empty partition block (Line 16). Thereafter, the existence of an empty partition block is checked (Line 17). These steps are repeated until no empty partition block exists (Line 14 - Line17).

In the fourth phase, *distribution*, the remaining nodes of the *c-list* are distributed into partition blocks in *ascending* order, until the *c-list* becomes empty. The node having lowest cost is removed from the *c-list* (Line 20) and is assigned to the partition block having the lowest cost (Line 21) until no cost nodes are available at the *c-list*. This minimizes cost disparity between partition blocks.

procedure partition[] *initial-partitioning*(Tree *cost-tree,* int *part-size*)

```
1:    // phase 1 : initialize c-list and p-array
2:    c-list ← all children nodes of root-node of the cost-tree
3:    p-array ← create empty partition blocks as many as part-size
4:    // phase 2 : expand, if necessary
5:    while lengthOf(c-list) < sizeOf(p-array) do
6:       if c-list contains coupled node(s) then
7:          comp ← remove a coupled node having the highest cost from c-list
8:          c-list += expand(comp)   // add all children nodes of comp to c-list
9:       else
10:         return error("can't expand...")
11:      endif
12:   endwhile
13:   // phase 3 : fill empty partition blocks in 'descending order'
14:   while an empty partition block exists in p-array
15:      comp ← remove a node having the highest cost from c-list
16:      assignTo(comp, any empty partition block in p-array)
17:   endwhile
18:   // phase 4 : distribute nodes in c-list into partition blocks in 'ascending order'
19:   while c-list is not empty do
20:      comp ← remove a node having the lowest cost from c-list
21:      assignTo(comp, the partition block having the lowest cost)
22:   endwhile
23:   return p-array
24: endprocedure
```

Algorithm 2. Initial Partitioning Algorithm

The **Figure 9** shows an example of the initial partitioning process when the requested number of partition blocks is 5. In the example, initially, the component list, *c-list*, is populated with nodes having costs 2, 25, 3, and 30. Those nodes are child nodes of a root node in a cost tree. The cost tree is constructed from a DEVS hierarchical model by applying a cost measure, which counts the number of internal states of a model, to the model. Since the length of the *c-list* is less than the requested number of partition blocks, 5, the expansion phase is required. In the expansion phase, the coupled node having the highest cost, 30, is removed from the *c-list*, and is expanded to nodes having costs 16, 8, and 6. Thereafter, those expanded nodes are stored in the *c-list*. No more expansion occurs because the length of the bag is now larger than the number of partition blocks. In the filling phase, nodes having the cost 25, 16, 8, 6, and 3 are selected and assigned to empty partition blocks. In the distribution phase, the remaining node in the *c-list,* which has a cost 2, is assigned to the partition block having the lowest cost, 3. The initial partitioning now terminates because there are no nodes in the *c-list*. Thus, the initial partitioning result becomes {{25}, {16}, {8}, {6}, {2,3}} for the given cost tree and the number of partition blocks. Initial partitioning results for various partition block sizes are presented in **Figure 10**.

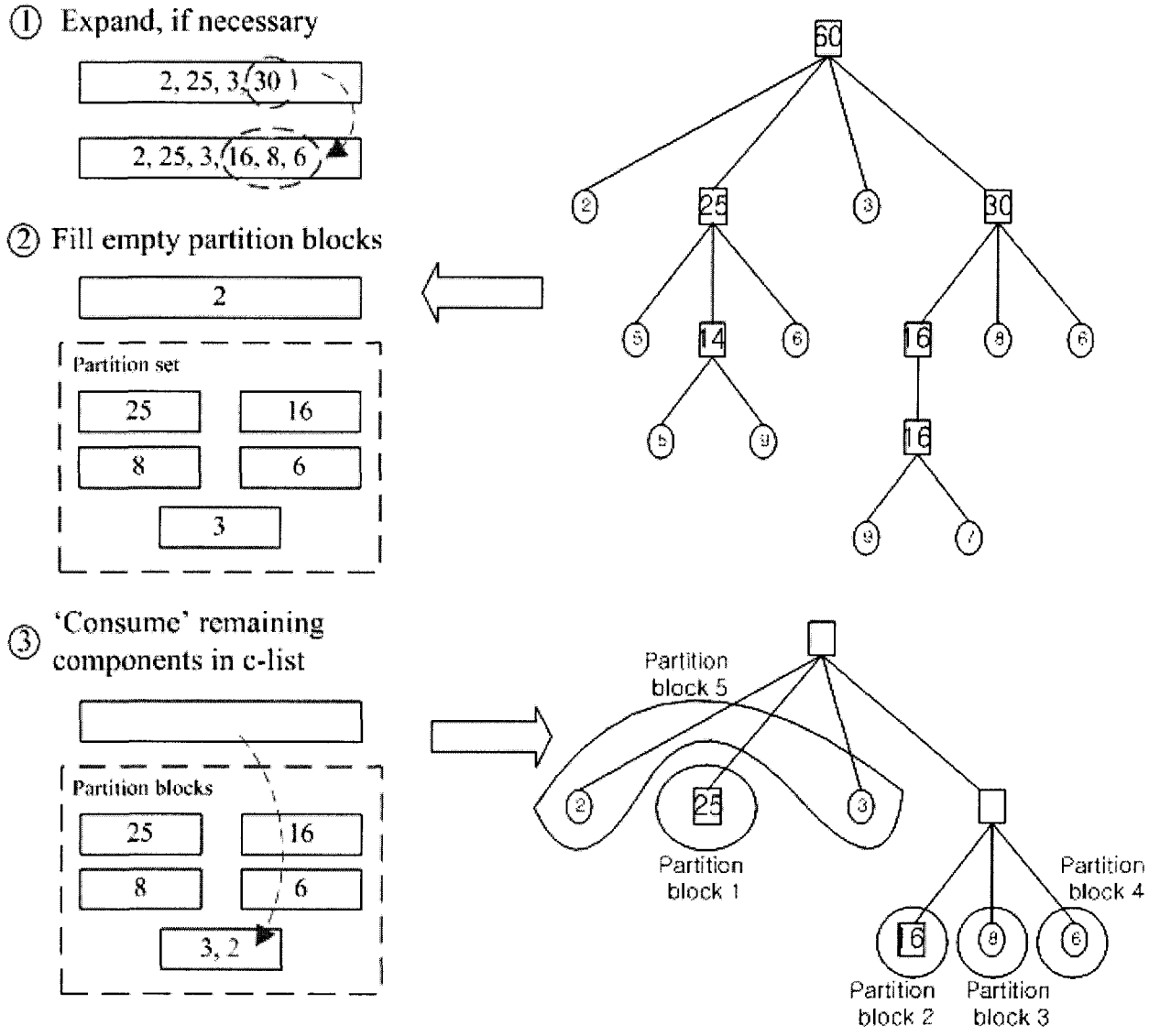


Figure 9. Initial Partitioning Process When the Number of Partition Blocks is 5

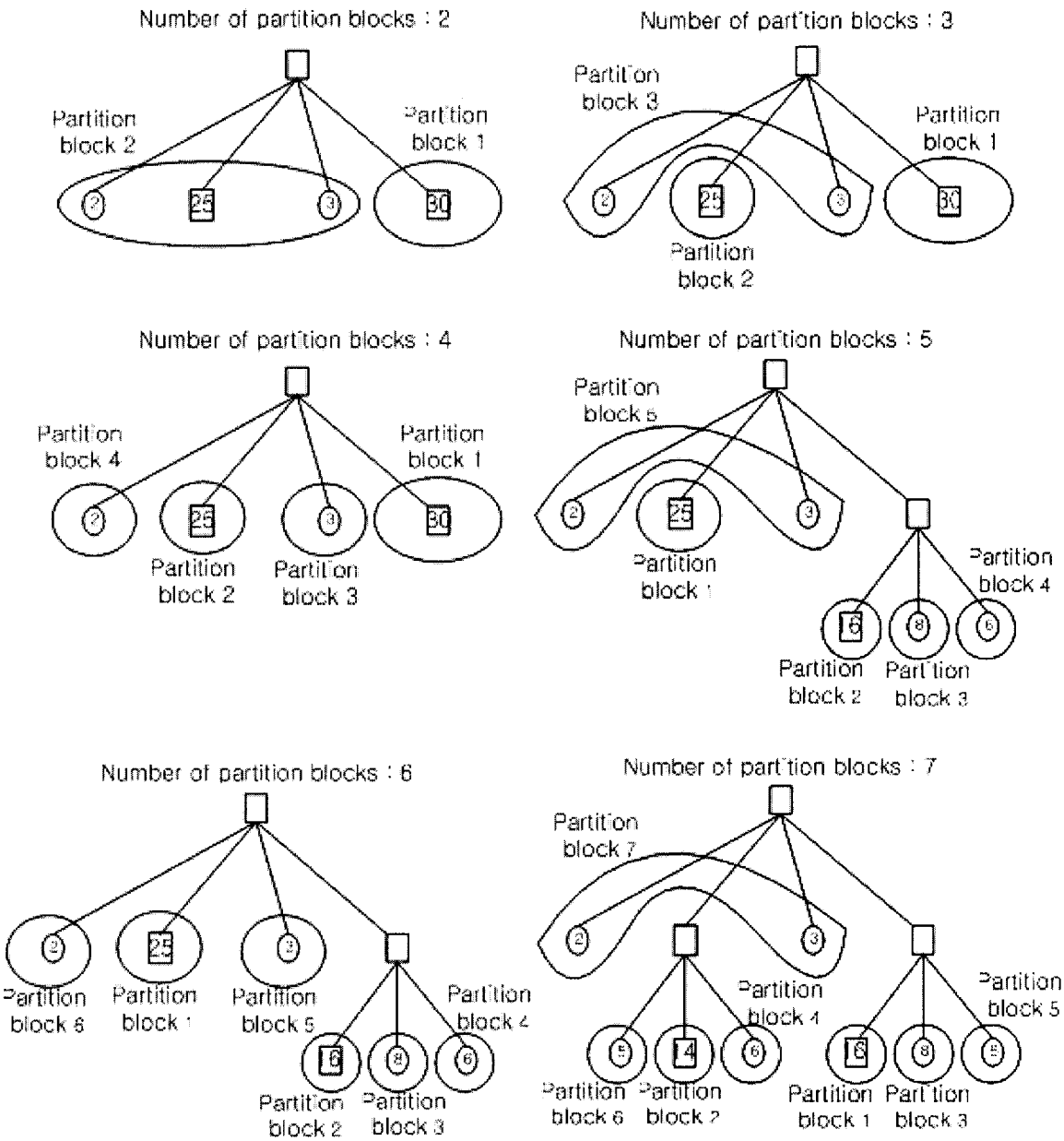


Figure 10. Initial Partitioning Results for Various Partition Block Sizes

4.2.2 Evaluation-Expansion-Selection (E^2S) Partitioning

Evaluation-Expansion-Selection (E^2S) partitioning constructs, evaluates, and expands child nodes of a partitioning tree and increases the QoP until a best partitioning result is attained. The E^2S partitioning is started by identifying the partition block having the highest cost given initial partitioning result that is stored at the root node of the partitioning tree. Once the block is identified as an expandable partition block, *e-partition*, the existence of a coupled model is checked at the block. If it exists, the coupled node having the highest cost is identified as an expandable node, *e-node*. Otherwise, a new partitioning block having the highest cost, excluding previously selected blocks, is selected as the expandable partition block. If a coupled node does not exist at the newly identified partition block, the same procedure is repeated until both an expandable block and an expandable node are found. By expanding the expandable node, a collection of cost nodes is created. If the expandable block becomes empty after expanding the node, the cost node having the highest cost in the collection is assigned to the block. Remaining nodes in the collection are distributed to partition blocks in ascending order, as described in distribution phase in the previous chapter. After distributing remaining nodes into partition blocks, the partitioning result is compared to the previous partitioning result. The cost measure, the partitioning evaluation function, is used to perform the cost comparison. If the new partition result is superior to the previous one, the E^2S partitioning is applied recursively to find a better partitioning result. Otherwise, the previous partitioning result is identified as a best partitioning result for the given cost tree and the requested number of partition blocks.

Similar to the initial partitioning algorithm, the E^2S partitioning algorithm is divided into six phases; *initialization*, *identification*, *expansion*, *filling*, *distribution*, and *evaluation*. In the first phase, *initialization*, three data objects, a collection of partition blocks for holding a new partitioning result, *e-array,* an expandable partition block, *e-partition*, and an expandable cost node, *e-node,* are created. The *e-array* is a collection of bags representing the E^2S partitioning result. It is replace by a better partitioning result, if one exists, as the algorithm proceeds recursively. The *e-partition* is the partition block having the highest cost in the *e-array* and the *e-node* is a cost node having the highest cost in the *e-partition.* In the algorithm, the *e-array* is assigned to the result of initial partitioning, *p-array* (Line 3). The *e-partition* and the *e-node* are initialized to the partition block having the highest cost in *e-array* (Line 4) and *null* (Line 5), respectively.

In the second phase, *identification*, the *e-partition* and the *e-node* are selected from the *e-array*. If the selected *e-partition* has no coupled node, another *e-partition* is chosen again from previously unselected partition blocks in the *e-array*. This is repeated until an *e-partition* containing a coupled node is found. If an *e-partition* having a coupled node is not found, the E^2S partitioning terminates by returning the previous partitioning result, *p-array*, as a best partitioning result. If the selected *e-partition* is empty, the algorithm terminates by returning the give *p-array* (Line 8). While the *e-partition* is not empty, the *e-node* is assigned to the node having the highest cost among coupled nodes in the *e-partition* (Line 10). If the *e-node* is not empty, the algorithm jumps to the next phase, expansion, by breaking out the while loop (Line 12). Otherwise, the algorithm terminates by returning the given *p-array* (Line 13). During this phase, the *e-node* is not always

selected from the partition block having the highest cost. If the *e-partition* does not contain any coupled node, it is reselected from *e-array* without any consideration of previously selected partition blocks (Line 16).

In the third phase, *expansion*, the expandable node, *e-node*, is removed from the *e-partition* and expanded. Thereafter, the component list, *c-list*, is created with expanded nodes. The *c-list* contains only child nodes of the *e-node*. The *c-list* in the initial partitioning algorithm holds all costs nodes available during the partitioning process. Thus, manipulation of the *c-list* is much simpler in the E^2S partitioning algorithm. The selected *e-node* is removed from the *e-partition* (Line 22). And, the *c-list* is constructed with expanded nodes of the *e-node* (Line 23).

In the fourth phase, *filling*, the *e-partition* is populated with a cost node if it becomes empty after expanding the *e-node*. The cost node is the node having the highest cost amongst the expanded nodes. In the E^2S partitioning, only *e-partition* is considered in the filling process because it is guaranteed that other partition blocks are not empty after the expansion phase. In the initial partitioning, all partition blocks are considered because they are all empty after the expansion phase. A cost node having the highest cost is removed from the *c-list* and assigned to the *e-partition* if the partition block is empty (Line26 – Line 28).

In the fifth phase, *distribution*, remaining cost nodes in the *c-list* are assigned to partition blocks, *e-array*, in ascending order. That is, a cost tree having the lowest cost is removed from the *c-list* and assigned to the partition block having the lowest cost until the *c-list* becomes empty. A cost node having the lowest cost is removed from the *c-list*

(Line 32), and the removed cost node is assigned to the partition block having the lowest cost (Line 33). These two steps are repeated until the *c-list* becomes empty (Line 31 – Line 34).

In the sixth phase, *evaluation*, the new partitioning result, *e-array*, is compared to the previous partitioning result, *p-array*. The comparison is conducted using a cost measure, the *partitioning evaluation function*. If the new result is superior to the previous one, a new partitioning node is created in the *e-array* and it becomes a child node of the partitioning node having the *p-array*. In this way, causality and QoP improvement are preserved between the child node and its parent node. Thereafter, the E^2S partitioning is recursively applied to find a better partitioning result based on the *e-array*. The new partitioning result, *e-array*, is compared to the previous partitioning result, *p-array* (Line 36). If the new result is better than the previous result, it recursively applies the algorithm again (Line 37). Otherwise, the E^2S partitioning algorithm terminates with the previous partitioning result, *p-array* (Line 39).

The Figure 11 depicts that how the E^2S partitioning works when the number of partition blocks is 5. In the example, the root node is equivalent to initial partitioning result shown in Figure 9. As a partitioning evaluation function, the cost function F, which computes cost disparity between a max partition block and a min partition block, is adopted. The function returns zero when all partition blocks are identical in terms of cost value. Otherwise, the function returns a positive value. Smaller value represents a better partitioning result.

```
1:  procedure partition[] e-square-s-partitioning(partition[] p-array)
2:    // phase 1: initialize e-array and e-partition
3:    e-array ← p-array
4:    e-partition ← a partition block having the highest cost in e-array
5:    e-node ← null
6:    // phase 2: identify an expandable partition block from e-array
7:    while true do
8:      if e-partition ≡ null then return p-array
9:      else
10:       if e-partition contains coupled node(s) then
11:         e-node ← the coupled node having the highest cost in e-partition
12:         if e-node ≠ null then break
13:         else return p-array
14:         endif
15:       else
16:         e-partition ← select the partition block having the highest cost from
17:                       e-array excluding previously selected partition blocks
18:       endif
19:     endif
20:   endwhile
21:   // phase 3: expand e-node and put them into c-list
22:   comp ← remove e-node from e-partition
23:   c-list ← expand(comp)
24:   // phase 4: fill the e-partition with the node having the highest cost
25:   // if the partition block is empty
26:   if empty(e-partition) then
27:     comp ← remove the node having the highest cost in c-list
28:     assignTo(comp, e-partition)
29:   endif
30:   // phase 5: distribute nodes to e-array
31:   while c-list is not empty do
32:     comp ← remove a node having the lowest cost from c-list
33:     assignTo(comp, a partition of e-array having the lowest cost)
34:   endwhile
35:   // phase 6: evaluate a new partitioning result
36:   if superiorTo(evaluate(e-array), evaluate(p-array)) then
37:     return e-square-p-partitioning(e-array)
38:   else
39:     retrun p-array
40:   endif
41: endprocedure
```

Algorithm 3. Evaluation-Expansion-Selection (E^2S) Partitioning Algorithm

The root node, $F(n_0)$, is evaluated by subtracting 5 from 25 because the max partition block and the min partition block contains {25} and {2, 3}, respectively. Similarly, nodes, n_1, n_2, and n_3 are respectively evaluated by subtracting {8} from {16}, {9} from {8, 7}, and {9} from {8, 7}. Since the disparity of the node n_3 is larger than the counterpart of the node n_2 the algorithm returns the collection of partition blocks that the node, n_2, contains as the final partitioning result. That is, {{14}, {9}, {8,7}, {6,6}, {2,3,5}}. The right-side partitioning nodes of each branch, (i.e., n_4 and n_5 in the example) demonstrate the possible partitioning result when the second highest partition block is selected and expanded. But, its partitioning result is always inferior to the partitioning result when the highest partition block is selected and expanded. That is, $F(n_4) \geq F(n_1)$ and $F(n_5) \geq F(n_2)$. The tree also shows that the disparity is improved as the depth of the tree increases (i.e., $20 \Rightarrow 8 \Rightarrow 6$). Similar to Figure 10, Figure 12 shows E^2S partitioning results for various partition sizes.

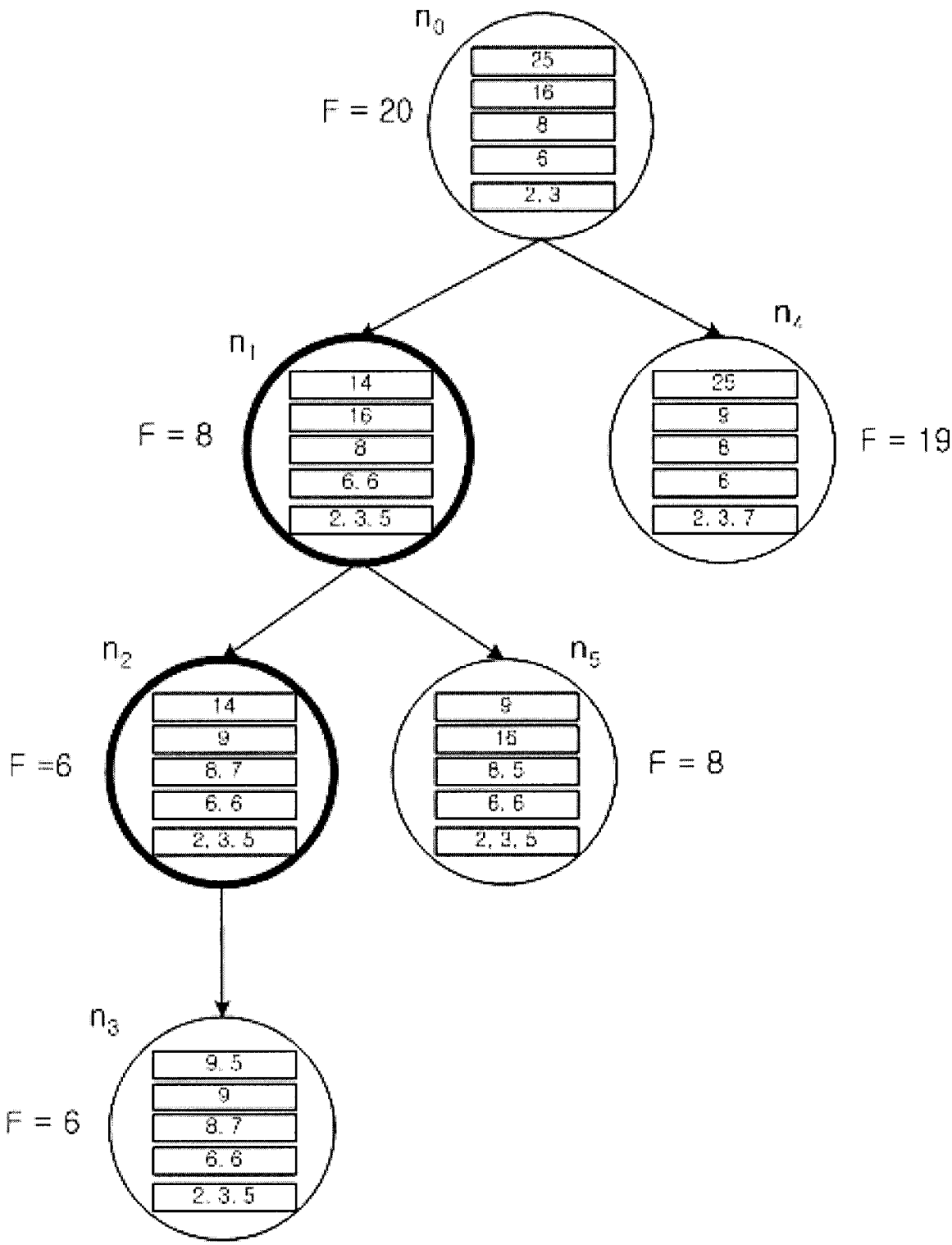


Figure 11. E^2S Partitioning Process When the Number of Partition Blocks is 5

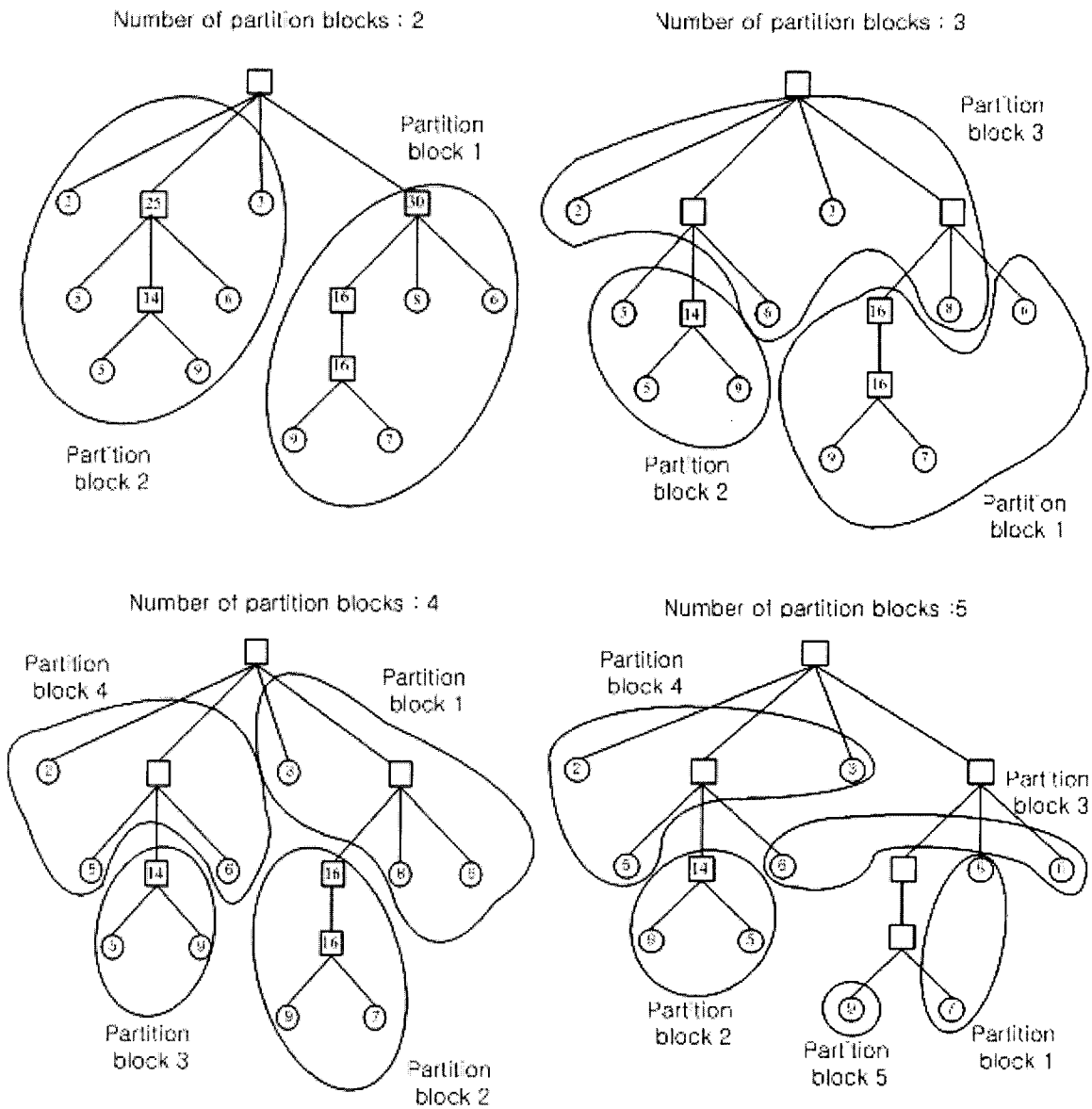


Figure 12. E^{2}S Partitioning Results for Various Partition Block Sizes

4.3 Algorithm Analysis

4.3.1 Initial Partitioning Algorithm

As described in the Algorithm 2, the component list, *c-list*, is initially populated with child nodes of the root node in a cost tree. Thus, the initial length of the *c-list*, l_0, is equivalent to the number of child nodes. As the cost tree is expanded, the length of the list is represented by

$$l_i = l_{i-1} - 1 + \xi_i, \quad i \geq 1 \qquad (1)$$

where, l_i = the length of the component list at i^{th} expansion
ξ_i = the number of children nodes of the expanded node at i^{th} expansion

When node expansion occurs, a coupled node is selected and removed from the component list, *c-list*. After the coupled node is expanded, its child nodes are added into the list. Thus, the length of the list at i^{th} expansion, l_i, becomes $l_{i-1} - 1 + \xi_i$.

$$\begin{aligned}
l_0 &= \xi_0 \\
l_1 &= l_0 - 1 + \xi_1 = \xi_0 + \xi_1 - 1 \\
l_2 &= l_1 - 1 + \xi_2 = (\xi_0 + \xi_1 - 1) - 1 + \xi_2 = \xi_0 + \xi_1 + \xi_2 - 2 \\
l_3 &= l_2 - 1 + \xi_3 = (\xi_0 + \xi_1 + \xi_2 - 2) - 1 + \xi_3 = \xi_0 + \xi_1 + \xi_2 + \xi_3 - 3 \\
&\vdots \\
l_{i-1} &= l_{i-2} - 1 + \xi_{i-1} = (\xi_0 + \xi_1 + \ldots + \xi_{i-2} - (i-2)) - 1 + \xi_{i-1} = \xi_0 + \xi_1 + \ldots + \xi_{i-1} - (i-1) \\
l_i &= l_{i-1} - 1 + \xi_i = (\xi_0 + \xi_1 + \ldots + \xi_{i-1} - (i-1)) - 1 + \xi_i = \xi_0 + \xi_1 + \ldots + \xi_{i-1} + \xi_i - i
\end{aligned}$$

By replacing l_{i-1} by the sum of ξ *up to i-1*th expansions, we can rewrite l_i as follows

$$
\begin{aligned}
l_i &= l_{i-1} - 1 + \xi_i \\
&= \sum_{j=0}^{i-1} (\xi_j - 1) - 1 + \xi_i \\
&= (\sum_{j=0}^{i-1} \xi_j + \xi_i) - (\sum_{j=0}^{i-1} 1 + 1) \\
&= \sum_{j=0}^{i} \xi_j - (i - 1 + 1) \\
&= \sum_{j=0}^{i} \xi_j - i
\end{aligned}
\qquad (2)
$$

The *while* loop in the phase of *expansion* is represented by

$$
while(l < p) = while((\sum_{i=0}^{N} \xi_i - N) < p) \qquad (3)
$$

where, l = the total length of the component list, *c-list*
p = the number of partition blocks, *p-array*
ξ_i = the number of children nodes of the expanded node at i^{th} expansion
N = total number of expansions

The total number of comparisons in the *while* loop is equivalent to the total number of expansions, N. The ξ_i can be formalized by a discrete random variable, $\boldsymbol{\xi}$, mapping the outcome in the sample space, i, into a positive integer in the real line, ξ_i, as shown in the **Figure 13**. Thus, by controlling the random variable, we can model and analyze various (or arbitrary) tree topologies. Nevertheless, to make analysis of the suggested algorithm simple, we adopt a constant random variable, **K**, mapping every outcome in the sample space into a constant value, k, in the real line. The k is a positive integer.

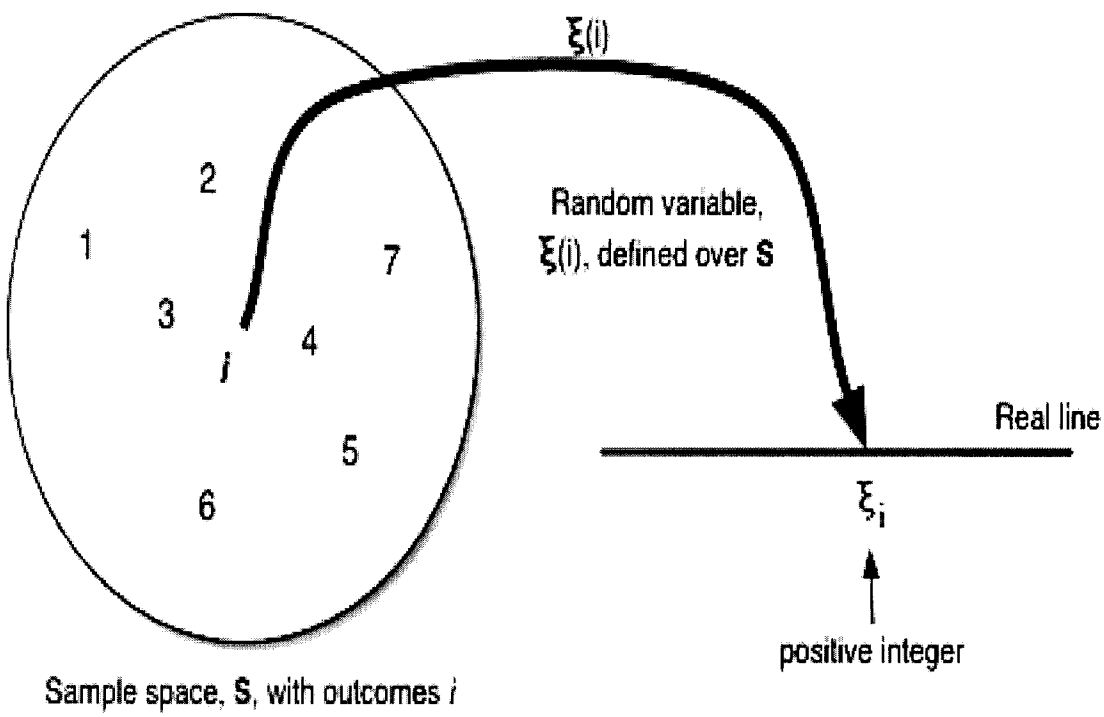


Figure 13. A Random Variable, **ξ,** Mapping the Outcome, *i*, to a Discrete Value, ξ_i

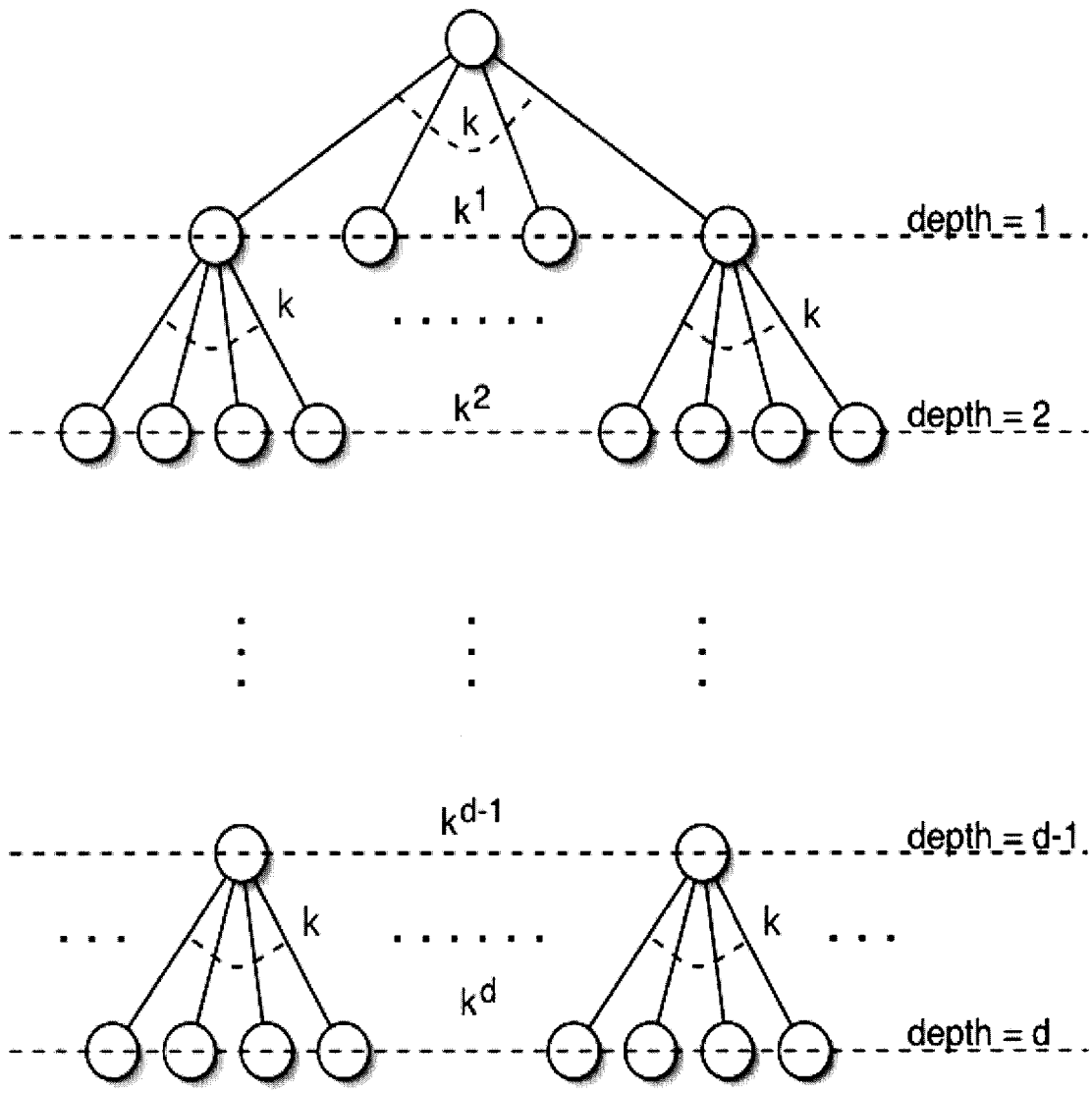


Figure 14. A Tree with the Depth *d*, and the Number of Children of a Coupled Node, *k*

For a given constant, k, the conditional statement in equation 3 is revised as follows

$$\begin{aligned}
&\left(\sum_{i=0}^{N} k - N\right) < p \\
&= ((N+1)k - N) < p \\
&= ((k-1)N + k) < p \\
&= (k-1)N < (p-k) \\
&= N < \frac{p-k}{k-1}, \qquad 1 < k < p
\end{aligned} \tag{4}$$

Since k is a constant, we can compute the total number of expansions, N, for a given number of partition blocks. The total number of comparisons, N, is defined by

$$N = \left\lceil \frac{p-k}{k-1} \right\rceil, \quad 1 < k < p \tag{5}$$

If the k is not smaller than the number of partition blocks, p, or is equal to 1, no comparisons happen. Otherwise, N comparisons occur. After the phase of *expansion*, it is guaranteed that the length of component list, l, is not smaller than the number of partition blocks, p. In the *filling* phase, p comparisons occur in the *while* loop because every empty partition in the *p-array* is filled with a cost node that is extracted from the *c-list*. (l-p) comparisons occur in the *distribution* phase because remaining nodes of the *c-list* are distributed into the *p-array*.

Definition 3:

n_{root}: the root node of a cost tree
$n_{highest}$: the node having the highest cost
n_{lowest}: the node having the lowest cost
p: the requested number of partition blocks
η: total number of expansions
$\xi(node)$: expanded nodes from a node, *node*
$\delta(c\text{-}list_{add}, nodes)$: time required for adding *nodes* to the *c-list*
$\delta\ (c\text{-}list_{remove}, nodes)$: time required for removing *nodes* from the *c-list*
$\delta\ (p\text{-}array_{create}, size)$: time required for creating the *p-array* with *size* empty blocks

$\delta(\xi(n_{selected}))$: time required for expanding the selected node, $n_{selected}$
δ ($partitioning_{initial}$): time required for performing the initial partitioning

The total execution time of the algorithm, $\delta(partitioning_{initial})$, is the summation of execution time of each step in the algorithm. That is,

$$\begin{aligned}&\delta(initialize) + \delta(\exp and) + \delta(fill) + \delta(distribute)\\ &= \delta(initialize) + \sum_{i=0}^{\eta}\delta(\exp and_i) + \sum_{i=1}^{p}\delta(fill_i) + \sum_{i=1}^{l-p}\delta(distribute_i)\end{aligned} \quad (6)$$

where, $\delta(initialize) = \delta\ (c\text{-}list_{add}, \xi(n_{root})) + \delta\ (p\text{-}array_{create}, p)$
$\delta(expand_i) = \delta\ (c\text{-}list_{remove}, n_{selected}) + \delta(\xi(n_{selected})) + \delta\ (c\text{-}list_{add}, \xi(n_{selected}))$
$\delta(fill_i) = \delta\ (c\text{-}list_{remove}, n_{highest}) + \delta\ (partition_{empty}, n_{highest})$
$\delta(distribute_i) = \delta\ (c\text{-}list_{remove}, n_{lowest}) + \delta\ (partition_{lowest}, n_{lowest})$

For the k-ary tree, $\delta(partitioning_{initial})$ is represented by

$$\begin{aligned}&\delta(initialize) + \sum_{i=1}^{\left\lceil\frac{p-k}{k-1}\right\rceil}\delta(\exp and_i) + \sum_{i=1}^{p}\delta(fill_i) + \sum_{i=1}^{l-p}\delta(distribute_i), && 1 < k < p\\ &\delta(initialize) + \sum_{i=1}^{p}\delta(fill_i) + \sum_{i=1}^{l-p}\delta(distribute_i), && k = 1 \vee k \geq p\end{aligned} \quad (7)$$

When we assume $\delta(fill_i)$ and $\delta(distribute_i)$ require the same execute time, the above equation becomes

$$\begin{aligned}&\delta(initialize) + \sum_{i=1}^{\left\lceil\frac{p-k}{k-1}\right\rceil}\delta(\exp and_i) + \sum_{i=1}^{l}\delta(assign_i), && 1 < k < p\\ &\delta(initialize) + \sum_{i=1}^{l}\delta(assign_i), && k = 1 \vee k \geq p\end{aligned} \quad (8)$$

where, $\delta(assign_i) = \delta(fill_i) = \delta(distribute_i)$

To simplify the analysis of this algorithm, we replace every δ $(\cdot)$ by the unit time, 1. Then, for given p and k, the above equation is simplified to

$$
\begin{aligned}
& 1+\left\lceil \frac{p-k}{k-1} \right\rceil + l \\
& = 1+\left\lceil \frac{p-k}{k-1} \right\rceil + \left((k-1)\cdot \left\lceil \frac{p-k}{k-1} \right\rceil + k \right) \\
& = k\cdot \left\lceil \frac{p-k}{k-1} \right\rceil + 1 + k \\
& = k\cdot \left(\left\lceil \frac{p-k}{k-1} \right\rceil + 1 \right) + 1
\end{aligned}
\qquad 1<k<p \qquad (9)
$$

The $\delta(partitioning_{initial})$ becomes k when $k = 1$ or $k \geq p$. Equation 9 implies that the total execution time of the initial partitioning algorithm is effected by the number of child nodes of each coupled node, k, rather than the size or the complexity of a given hierarchical model. Indirectly, performance of the algorithm is related to the number of expansions, N, rather than model complexity.

To conduct worst case analysis of the algorithm, we define a cost tree as follows

Definition 4:

T(*d, k, n*): A cost tree that consists of the depth, d, the number of child nodes of a coupled node, k, and the total number of atomic nodes in the tree, n.

The worst case occurs when partitioning a cost tree, T(d, k, n), that satisfies the following conditions

i) the number of partition blocks, p, is k^d,
ii) the number of expansions, e, is k^{d-1},
iii) the total number of atomic nodes, n, is k^d,
iv) $k << p$. That is, k is 2 and p is k^d

By applying the above constraints to equation 8, we get

$$\delta(initialize) + \sum_{i=1}^{k^{d-1}} \delta(expand_i) + \sum_{i=1}^{k^d} \delta(assign_i) = 1 + k^{d-1} + k^d \qquad (10)$$

From above conditions, we can rewrite d and k as follows

$$k^d = n \Rightarrow d = \log_k n$$
$$\log_e k^d = \log_e n \Rightarrow d \cdot \log_e k = \log_e n \Rightarrow \log_e k = \tfrac{1}{d} \cdot \log_e n \Rightarrow k = e^{\frac{1}{d} \cdot \log_e n} \Rightarrow k = n^{\frac{1}{d}} \quad (11)$$

By integrating d and k in equation 11 with equation 10, The total execution time of the algorithm in the worst case for the given cost tree, T(d, k, n), is computed as follows

$$\begin{aligned} & 1 + k^{d-1} + k^d \\ &= 1 + k^{\log_k n - 1} + k^{\log_k n} \\ &= 1 + \frac{n^{\log_k k}}{k} + n^{\log_k k} \\ &= 1 + \frac{1}{k} \cdot n + n \\ &= 1 + n^{-\frac{1}{d}} \cdot n + n \\ &= 1 + n^{1-\frac{1}{d}} + n \end{aligned} \quad (12)$$

In the worst case, the execution time of the initial partitioning is represented by $\Theta(n)$ for a given cost tree, T(d, k, n), because $\Theta(1 + n^{1-\frac{1}{d}} + n)$ is equivalent to $\Theta(n)$ when $d > 1$.

4.3.2 E^2S Partitioning Algorithm

The algorithm has six phases as described in Algorithm 3. Thus, the total execution time of the algorithm, $\delta(E^2S - partitioning)$, is represented by

$$\delta(initialize) + \delta(identify) + \delta(\exp and) + \delta(fill) + \delta(distribute) + \delta(evaluate) \quad (13)$$

Definition 5:

$\xi_{\text{e-node}}$: the number of child nodes expanded from *e-node*, $\xi(node_{\text{e-node}})$
n_{highest}: the node having the highest cost
n_{lowest}: the node having the lowest cost
γ : the number of recursions until a best partitioning result is attained
$\delta(e\text{-}array_{\text{create}}, partitions)$: time required for creating the *e-array* with *partition blocks.*
$\delta(e\text{-}array_{\text{select}}, partition)$: time required for finding the partition block, *partition*, from the *e-array* excluding previously selected partition blocks.
$\delta(e\text{-}partition_{\text{elect}}, node)$: time required for finding the node, *node*, from the *e-partition.*

In *identification* phase, the total number of comparisons in the w*hile* loop, r, ranges from 1 to the number of partition blocks, p, depending on the existence of a coupled node in the expandable partition, *e-partition.* This is because a new *e-partition* is selected $(r\text{-}1)$ times before identifying the *e-partition* having at least one coupled node. Time for identifying the *e-partition* from the *e-array*, $\delta(identify)$, is represented by

$$\sum_{i=1}^{r} \delta(identify_i) = \sum_{i=1}^{r-1} \delta(identify_i, e-array) + \delta(identify, e-node) \quad (14)$$

where, r : the number of iterations for identifying the *e-partition* having a coupled node from the *e-array*
$\delta(identify_i, e-array) : \delta(e-array_{select}, e-partition)$
$\delta(identify, e-node) : \delta(e-partition_{select}, e-node)$

The component list, *c-list*, is populated with child nodes of the *e-node* in the *expansion* phase and its length decreases by 1 in the *filling* phase if the *e-partition*

becomes empty after expanding the *e-node*. Time required for expanding the *e-node*, δ(*expand*), is represented by

$$\delta(e-partition_{remove}, e-node) + \delta(c-list_{init}, (\xi(e-node)) \quad (15)$$

Similarly, the time required for filling an empty *e-partition*, δ(*fill*), is

$$\delta(c-list_{remove}, n_{highest}) + \delta(e-partition, n_{highest}), \text{ iff } e\text{-}partition \text{ is empty} \quad (16)$$

In *distribution* phase, after expanding *e-node*, the length of the *c-list*, *l*, is equivalent to the number of child nodes of the *e-node*, $\xi_{\text{e-node}}$, if the *e-partition* is not empty. Otherwise, *l* becomes $\xi_{\text{e-node}} - 1$. Thus, the time required for distributing nodes of the *c-list* into the *e-array*, δ(*distribute*), is represented by

$$\delta(distribute) = \sum_{i=1}^{\xi_{e-node}-\varepsilon} \delta(distribute_i) \quad (17)$$

where, $\delta(distribute_i) = \delta(c\text{-}list_{\text{remove}}, n_{\text{lowest}}) + \delta(partition_{\text{lowest}}, n_{\text{lowest}})$

ε : filling flag. If filling happens, ε is 1. Otherwise, zero.

The time required for evaluating a new partitioning result and recursively finding a better result in the phase of *evaluation*, δ(*evaluate*), is

$$\gamma \cdot \left(\delta(initialize) + \delta(identify) + \delta(\exp and) + \delta(fill) + \delta(distribute)\right) \quad (18)$$

where, γ : the number of recursions until a best partitioning result is achieved

Equitation 18 is only valid when a new partitioning result is superior to an old result in terms of cost. By substituting equations from 14 to 17 into equation 13, we can obtain the total execution time of the algorithm, $\delta(E^2S\text{-}partitioning)$ as follows

$$\begin{aligned}
&\delta(initialize) + \delta(identify) + \delta(\exp and) + \delta(fill) + \delta(distribute) + \delta(evaluate) \\
&= \delta(initialize) + \sum_{i=1}^{r} \delta(identify_i) + \delta(\exp and) + \varepsilon \cdot \delta(fill) + \sum_{i=1}^{\xi_{e-node}-\varepsilon} \delta(distribute_i) + \delta(evaluate) \\
&= \delta(initialize) + r \cdot \delta(identify_i) + \delta(\exp and) + \varepsilon \cdot \delta(fill) + \left(\xi_{e-node} - \varepsilon\right) \cdot \delta(distribute_i) + \delta(evaluate) \quad (19)
\end{aligned}$$

We assume $\delta(fill)$ and $\delta(distribute_i)$ require the same amount of time to perform their operations. Then, equation 19 can be rewritten as

$$\delta(initialize) + r \cdot \delta(identify_i) + \delta(\exp and) + \xi_{e-node} \cdot \delta(assign_i) + \delta(evaluate) \quad (20)$$
$$\text{where, } \delta(assign_i) = \delta(fill) = \delta(distribute_i)$$

To simplify the analysis, we replace every $\delta(\cdot)$, except for $\delta(evaluate)$, by the unit time, 1. Then, the total execution time of the algorithm, $\delta(E^2S\text{-}partitioning)$, becomes

$$\begin{aligned} & 1 + r \cdot 1 + 1 + \xi_{e-node} \cdot 1 + \delta(evaluate) \\ & = 2 + r + \xi_{e-node} + \delta(evaluate) \end{aligned} \quad (21)$$

By substituting equation 18 into equation 21, we get

$$2 + r_0 + \xi^0_{e-node} + \sum_{i=1}^{\gamma}\left(2 + r_i + \xi^i_{e-node}\right) = \sum_{i=0}^{\gamma}\left(2 + r_i + \xi^i_{e-node}\right) \quad (22)$$
where, ξ^i_{e-node} : ξ_{e-node} when expansions occurs i times

In equation 22, r_i is the number of iterations needed to identify the *e-partition* having a coupled node at the i^{th} expansion. By replacing the summation of r_i by an average, $\bar{r}$, we can rewrite the equation 22 as follows

$$\sum_{i=0}^{\gamma}\left(2 + \bar{r} + \xi^i_{e-node}\right) \quad (23)$$
$$\text{where, } \bar{r} = \sum_{i=0}^{\gamma} \frac{r_i}{\gamma + 1}$$

For a k-ary tree, ξ^i_{e-node} is k. Then, equation 23 can be rewritten as

$$\sum_{i=0}^{\gamma}\left(2 + \bar{r} + k\right) = (\gamma + 1) \cdot (2 + \bar{r} + k) \quad (24)$$

Equation 24 implies that the total execution time of the E^2S partitioning algorithm, $\delta(E^2S - partitioning)$, is affected by the degree of QoP rather than the size or the complexity of the given hierarchical model.

The worst case occurs when partitioning the cost tree, T(*d, k, n*), that satisfies the following conditions

i) No expansion occurs in the initial partitioning. That is, $k \geq p$
ii) Every coupled node expands
iii) A new partitioning result is always superior to the previous result

By applying the above constraints to equation 24 and replacing $\bar{r}$ by *1*, we get

$$(\gamma + 1) \cdot (2 + 1 + k) \qquad (25)$$

In the worst case, every node up to the *d-1* layer is a coupled node. This implies that a coupled node exists in the max partition block during the partitioning process. Thus, $\bar{r}$ is becomes *1*. In general, $\bar{r}$ varies between 1 and p during the partitioning process. Based on the second condition, we can compute γ for the given cost tree, T(*d, k, n*), as follows

$$\gamma = \sum_{i=1}^{d-1} k^i = k \cdot \left(\frac{k^{d-1} - 1}{k - 1} \right), \qquad k > 1 \qquad (26)$$

The γ is equal to the summation of the total number of coupled nodes except for the root node. By integrating γ in equation 26 with equation 25, the total execution time of the E^2S algorithm in the worst case for a given cost tree, T(*d, k, n*), is computed as follows

$$
\begin{aligned}
&\sum_{i=0}^{\gamma}\left(2+\bar{r}+\xi_{e-node}^{i}\right) \\
&=\sum_{i=0}^{\gamma}(2+1+k) \\
&=(\gamma+1)\cdot(3+k) \\
&=\left(\left(k\cdot\frac{\left(k^{d-1}-1\right)}{(k-1)}\right)+1\right)\cdot(3+k) \\
&=\left(\left(\frac{k^{d}-k}{k-1}\right)+1\right)\cdot(3+k) \qquad (27) \\
&=\left(\left(\frac{n^{\frac{1}{d}\cdot d}-n^{\frac{1}{d}}}{n^{\frac{1}{d}}-1}\right)+1\right)\cdot\left(3+n^{\frac{1}{d}}\right) \\
&=\left(\frac{n-n^{\frac{1}{d}}+n^{\frac{1}{d}}-1}{n^{\frac{1}{d}}-1}\right)\cdot\left(3+n^{\frac{1}{d}}\right) \\
&=\frac{(n-1)\cdot\left(3+n^{\frac{1}{d}}\right)}{n^{\frac{1}{d}}-1} \\
&\approx n
\end{aligned}
$$

In the worst case, the execution time of the E^{2}S partitioning is represented by $\Theta(n)$ for a cost tree, T(d, k, n), because $\Theta\left(\frac{(n-1)\cdot\left(3+n^{\frac{1}{d}}\right)}{n^{\frac{1}{d}}-1}\right)$ is equivalent to $\Theta(n)$ when $d > 1$.

4.4 Optimality Issues

4.4.1 Optimal Size of Partition Blocks

In the proposed algorithm, we assume that the total number of partition blocks is provided by a user. Based on the number of blocks, the algorithm produces a best partitioning result for a given hierarchical model. However, it cannot be optimal until the partitioning results of all possible partition block sizes are inferior to the best result. In the initial partitioning algorithm, we can find the optimal number of partition blocks by identifying the number of partition blocks, p, which satisfies the following equation

$$\begin{aligned} &\min_{p=1..\infty}\left\{\delta_p^K(initial-partitioning)\right\} \\ &= \min_{p=2..\infty}\begin{cases} K\cdot\left(\left\lceil\frac{p-K}{K-1}\right\rceil+1\right)+1, & 1<K<p \\ K, & K=1\vee K\geq p \end{cases} \end{aligned} \tag{28}$$

The optimal number of partition blocks, p_{opt}, satisfying the above equation is

$$p_{opt} = \begin{cases} K+1, & p>K>1 \\ p, & p\leq K \end{cases} \tag{29}$$

When p > K, the optimal number of partition blocks, p_{opt}, is K+1 because the execution time of the algorithm grows linearly as p increases. The p_{opt} is independent from K when p ≤ K. Thus, p_{opt} becomes p regardless of K. In the E^2S partitioning, the execution time of the algorithm is decided by the degree of QoP rather than the number of partition blocks for the given cost tree, T(d, k, n), as shown in equation 24. Thus, we cannot find the optimal number of partition blocks in the E^2S partitioning. It is also notable that E^2S partitioning is initiated from the initial partitioning result. That is, the number of partition blocks created in the initial partitioning process is not dynamically

changed during E^2S partitioning process. **Figure 16** shows p_{opt} for various K when $1 < K < p$.

4.4.2 Cost Tree Optimization for a Particular Partition Size

For a fixed number of partition blocks, P, the optimal number of k in a k-ary cost tree, k_{opt}, for minimizing total execution time of the initial partitioning algorithm is identified by finding an appropriate k satisfying the following equation

$$\min_{k=1..\infty}\left\{\delta_P^k(initial-partitioning)\right\} = \min_{k=2..P-1}\begin{cases} k\cdot\left(\left\lceil\frac{P-k}{k-1}\right\rceil+1\right)+1\Big\}, & 1<k<P \\ k, & k=1 \vee k \geq P \end{cases} \quad (30)$$

The optimal number of fan-outs of a coupled node, k_{opt}, is defined by

$$k_{opt} = \begin{cases} \left\lfloor\frac{P}{2}+1\right\rfloor, & 1<k<P \\ k, & k\equiv 1 \vee k \geq P \end{cases} \quad (31)$$

For the condition of $1 < k <$ P, we can find the k_{opt} making the total execution time of the initial partitioning minimal by applying various k, which range from 2 to P-1, into equation 30 along with the given number of partition blocks, P. The k_{opt} is k for the condition of $k = 1 \vee k \geq$ P. Figure 15 shows k_{opt} for various numbers of partition blocks when $1 < k <$ P. In the E^2S partitioning, the k_{opt} is 2 when the worst case analysis is considered. This is because the smallest k, which is greater than 1, produces the minimum execution time for the algorithm.

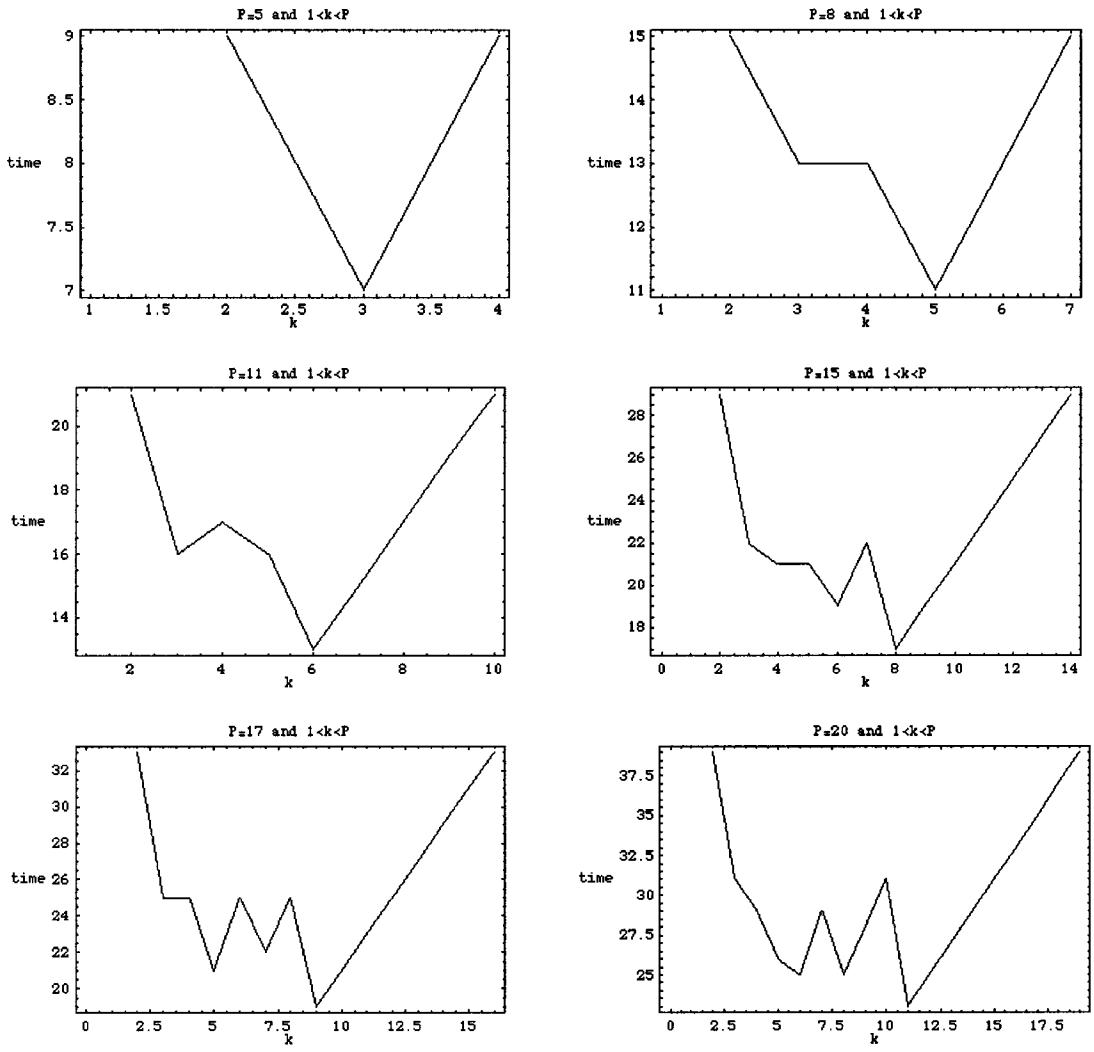


Figure 15. Finding Optimal *k* for Various Numbers of Partition Blocks when $1 < k < P$

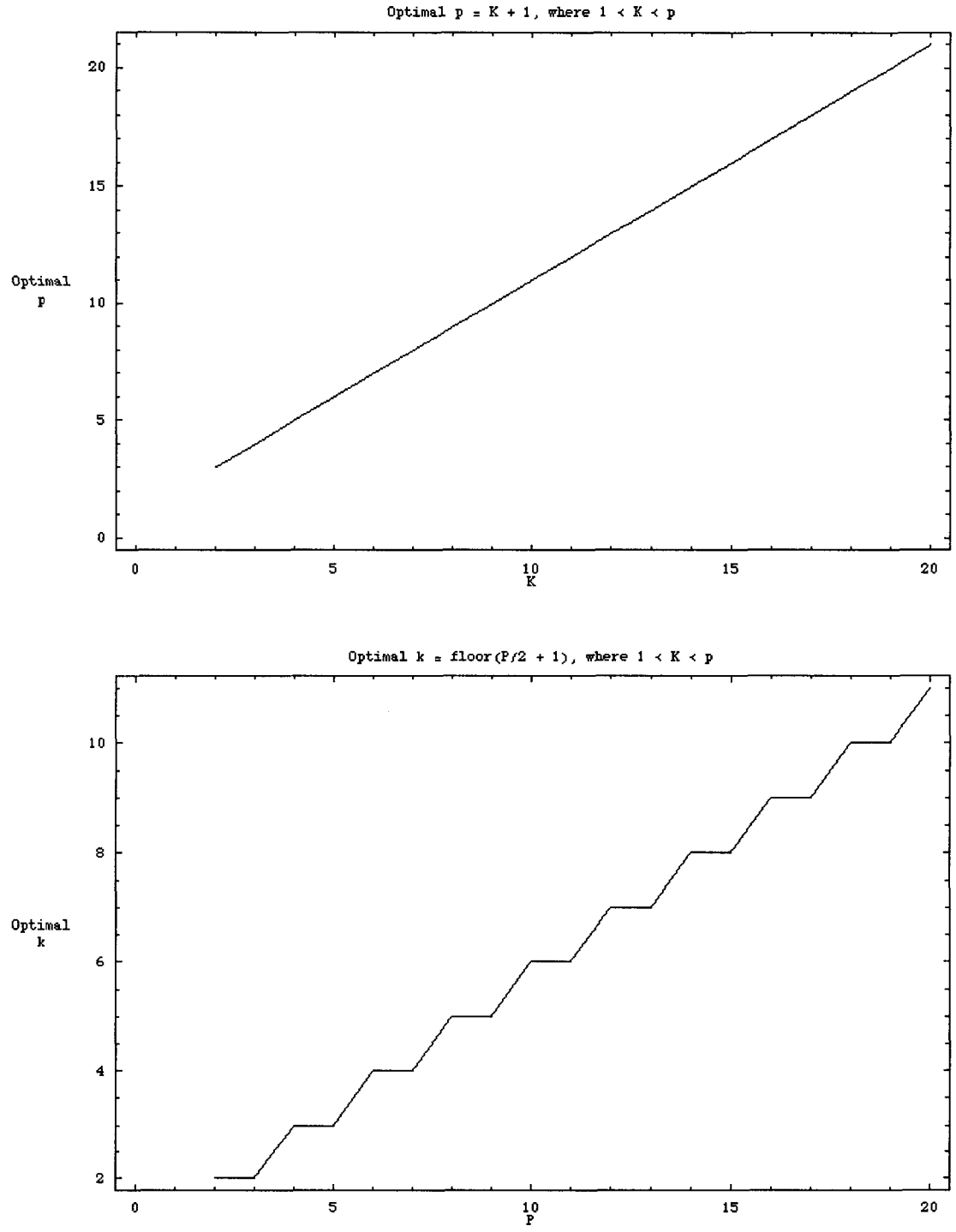


Figure 16. Finding p_{opt} and p_{opt} When $1 < k <$ p in the Initial Partitioning Process

CHAPTER 5. ADVANCED ALGORITHMS: EXTENDIBILITY AND ADAPTABILITY

5.1 Classification of GMP Algorithms

New partitioning algorithms derived from the GMP algorithm (*GMP-baseline* algorithm hereafter) are presented in this chapter. These algorithms are created to tackle numerous partitioning problems in various distributed computing environments. Extendibility and adaptability of the GMP-baseline algorithm is demonstrated by investigating the derived algorithms. These algorithms are categorized in **Table 4**.

Table 4. Classification of Generic Model Partitioning algorithms

	Static Information	Dynamic Information
Homogeneous System	GMP-SHM	GMP-DHM
Heterogeneous System	GMP-SHT	GMP-DHT

A generic model partitioning algorithm for static information/homogeneous system, GMP-SHM, is the algorithm introduced in the previous chapter. This algorithm assumes that all hosts (or logical simulators) are identical and each host provides unlimited resources (or costs). Time-varying information about hosts or resource is not considered during the partitioning process. The algorithm produces an identical partitioning result no matter how many times it is executed for a given model. A new GMP-SHM algorithm, *GMP with look-ahead (l)*, is introduced and described in this chapter to show the extendibility of the GMP algorithm. *Partitioning granularity* and *QoP enhancement* are

discussed regarding the *GMP with look-ahead (l)*. The class of GMP-SHM algorithms is also referred to as *blind partitioning algorithms*.

A generic model partitioning algorithm for static information/heterogeneous system, GMP-SHT, is a GMP algorithm that decomposes a hierarchical model into a set of partition blocks based on time-invariant resource capacity in a heterogeneous system. A host in the heterogeneous system is represented by a time-invariant resource capability, *total cost*, and is associated with a partition block. Unlike the GMP-SHM, which assumes every partition block to contain unlimited number of cost nodes regardless of their costs, the GMP-SHT forces a partition block to enclose cost nodes only until their aggregated cost becomes equal to or less than the total cost of its corresponding host. During the partitioning process, the *total cost* is considered before a cost node is assigned to a particular partition block. Similar to the GMP-SHM, the algorithm provides an identical partitioning result regardless of the number of times it is applied to the given model. This algorithm is also referred to a *host-aware partitioning algorithm.*

A generic model partitioning algorithm for dynamic information/homogeneous system, GMP-DHM, is a GMP algorithm that decomposes a hierarchical model into a set of partition blocks based on time-varying resource availability in a homogeneous system. A host in homogeneous system is abstracted by its time-varying resource availability, *available cost*, and is assigned to a partition block. The available cost of a host fluctuates through time depending on resource allocation by the host. Similar to the GMP-SHT, the GMP-DHM forces a partition block to accept cost nodes only until their aggregated cost is equal to or less than the available cost of its corresponding host. The upper-limit of the

aggregated cost that a partition block can hold also varies through time because it is directly influenced by available cost fluctuation. During the partitioning process, the available cost is considered before a cost node is assigned to a partition block. Unlike GMP algorithms based on static information (i.e., GMP-SHM and GMP-SHT), this algorithm may produce a different partitioning result when it is executed at some later time for a given model. This algorithm is also referred to a *resource-aware partitioning algorithm.*

A generic model partitioning algorithm for dynamic information/heterogeneous system, GMP-DHT, is a GMP algorithm that decomposes a hierarchical model into a set of partition blocks based on time-varying resource capability in a heterogeneous system. A host is abstracted by its time-varying resource capacity, *temporal total cost,* and is assigned to a partition block. The total cost of a host fluctuates through time depending on resource availability of the host at a specific time. Similar to the GMP-DHM, the GMP-DHT forces a partition block to accept cost nodes only until their aggregated cost is equal to or less than the *total cost* of its corresponding host. During the partitioning process, the total cost is considered before a cost node is assigned to a partition block. The total cost fluctuates through time depending on the activities of its corresponding host. This algorithm, similar to the GMP-DHM, may produce a different partitioning result when it is executed at some later time for a given model. This algorithm is also referred to as a *host-sensitive partitioning algorithm.*

5.2 GMP Algorithm with Look-ahead (l)

The GMP-baseline algorithm introduced in the previous chapter presumes that a partition block represents a host in homogeneous system. Thus, the algorithm performs the partitioning process to minimize cost disparity between partition blocks to achieve a high degree of QoP. Nevertheless, it considers neither cost disparity between cost nodes nor the partitioning granularity of the final result. When we are dealing with more sophisticated distributed system such as a network of clusters or a network of sub-networks, it is possible that a partition block should represent a set of hosts or a sub-network instead of a single host. In this scenario, it is desirable to maintain minimum cost disparity between cost nodes or to reduce partitioning granularity to obtain a better partitioning result for the given network infrastructure. Even though a partition block represents a single host, certain benefits are attained from cost disparity minimization and partitioning granularity reduction. For example, partitioning granularity reductions increases cost homogeneity between cost nodes. It allows the host to schedule or allocate cost nodes more efficiently. To cope with those issues, a new algorithm, *GMP with look-ahead(l)*, is proposed in this chapter.

A generic model partitioning algorithm with look-ahead(l), GMP-look-ahead(l), is a GMP algorithm that produces a better partitioning result in terms of cost disparity between cost nodes or partitioning granularity of the result, compared with the GMP-baseline algorithm, by considering a look-ahead coefficient, *l*. There are two schemes for managing the coefficient. These are *aggressive look-ahead* and *deferred look-ahead*. In the aggressive look-ahead scheme, in addition to atomic nodes in the *c-list*, all expanded

cost nodes up to l^{th} level from every coupled node in the list (during initial partitioning) or in partition blocks (during E^2S partitioning) are also assigned to partition blocks. Thus, the partitioning result may differ from the result created by the GMP-baseline algorithm. The algorithm produces a partitioning result having a smaller disparity between cost nodes compared to the GMP-baseline algorithm.

In the deferred look-ahead scheme, partitioning is conducted as in the GMP-baseline algorithm until a best partitioning result is attained. After the result is obtained, every coupled node in the result is expanded up to l^{th} level from the node. This reduces partitioning granularity of the result.

The proposed algorithm is easily constructed from the GMP-baseline algorithm by adding a phase for expanding coupled nodes up to l times just before assigning cost nodes to partition blocks (in the aggressive scheme) or just after obtaining the final partitioning result (in the deferred scheme). The rest of algorithm is equivalent to the GMP-baseline algorithm.

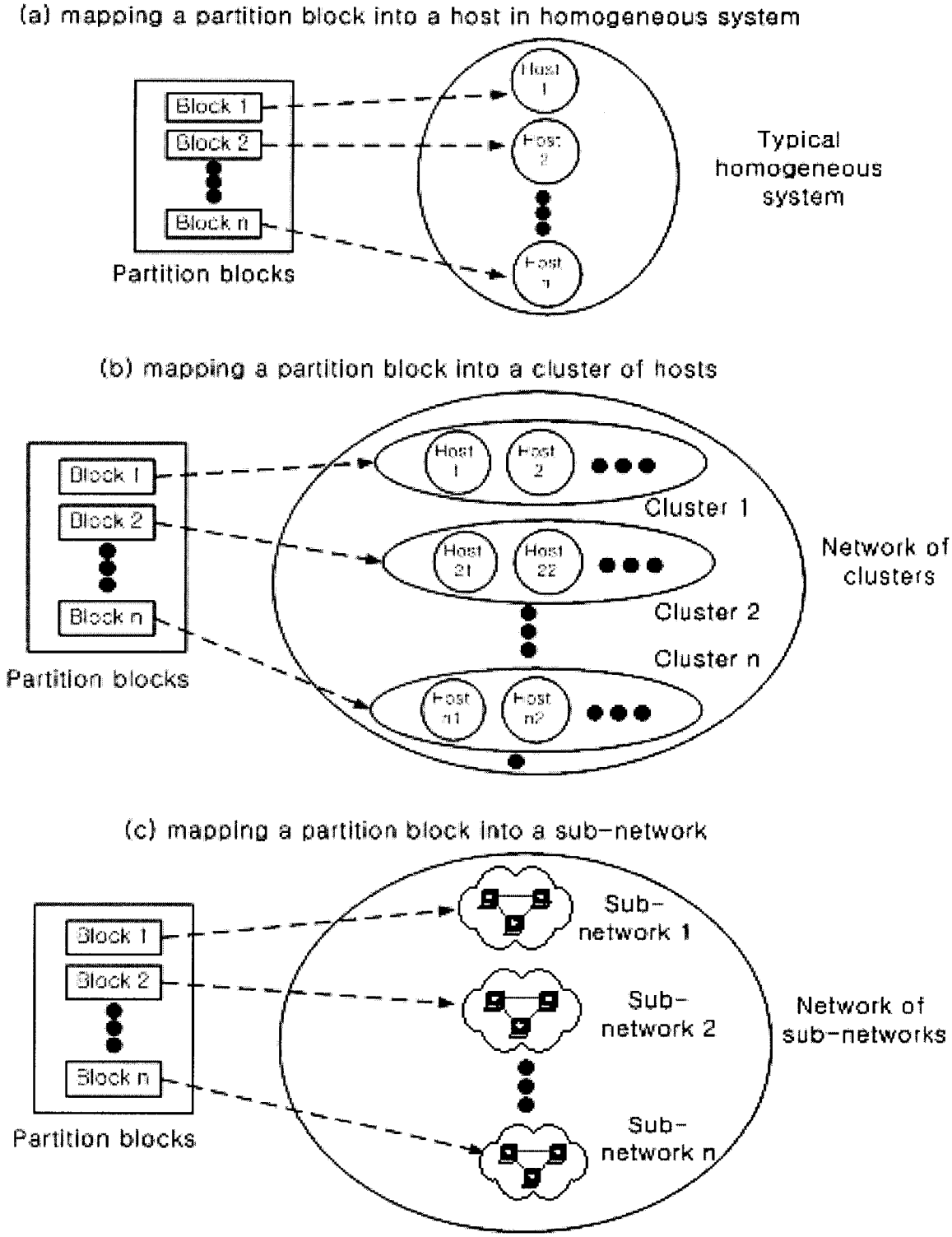


Figure 17. Partition Block Mapping in Various Distributed Network Systems

5.2.1 Initial Partitioning with Look-ahead (l)

Initial partitioning in the GMP with look-ahead (l) (*initial partitioning with look-ahead(l)*) requires a third parameter, *look-ahead coefficient*, l, in addition to the existing parameters; a cost tree and the number of partition blocks. The look-ahead parameter, l, represents a relative offset (or depth) from a cost node to another node that could be reachable via traversal of the cost tree. The relative offset is represented by the number of links between those nodes. A link is a direct connection between two nodes without any intermediate nodes. Initial partitioning algorithms of the GMP with look-ahead (l) for both the aggressive scheme and the deferred scheme are presented in Algorithm 4 and Algorithm 5, respectively. Partitioning results regarding those algorithms are illustrated in Figure 18 when the look-ahead coefficient is 2 and the number of partition blocks is 5.

Figure 18.a shows the initial partitioning result when the aggressive scheme is used. The result, {{9,7}, {8,6}, {6,5}, {2,3,5}, {9}}, differs from the counterpart generated by the GMP-baseline algorithm, {{25}, {16}, {8}, {6}, {2,3}}. This is because all coupled nodes in the *c-list* are expanded up to l times before assigning each cost node in the list to a particular partition block (Line 15 – Line 24 in **Algorithm 4**). The main advantage of this scheme is cost disparity reduction. Compared to the cost disparity of the partitioning result achieved by the GMP-baseline algorithm, 20, this algorithm produces smaller cost disparity, 7. Because cost disparity is highly correlated with the degree of QoP, the algorithm could produce a better alternative compared the partitioning result generated by the GMP-baseline algorithm.

Figure 18.b shows the initial partitioning result when the deferred scheme is used. The result, {5,5,6,9}, {9,7}, {8}, {6}, {2, 3}}, is identical to the result generated by the GMP-baseline algorithm, {{25}, {16}, {8}, {6}, {2,3}} in terms of cost disparity between the max- and the min-partition blocks. This is because all coupled nodes in the partition blocks are expanded up to *l* times after attaining a best partitioning result (Line 26 – Line 38 in **Algorithm 5**). The main advantage of this scheme is partitioning granularity reduction. The algorithm produces a fine-grain partitioning result while preserving identical cost disparity between partition blocks as compared to the result of the GMP-baseline algorithm.

Table 5. Characteristics of GMP with Look-ahead (l) Algorithm

	Aggressive scheme	Deferred scheme
Look-ahead Expansion	Before assigning cost nodes	After finding a partitioning result
Cost disparity between max- and min-partition block	reduced	invariant
Partitioning granularity	N/A	reduced

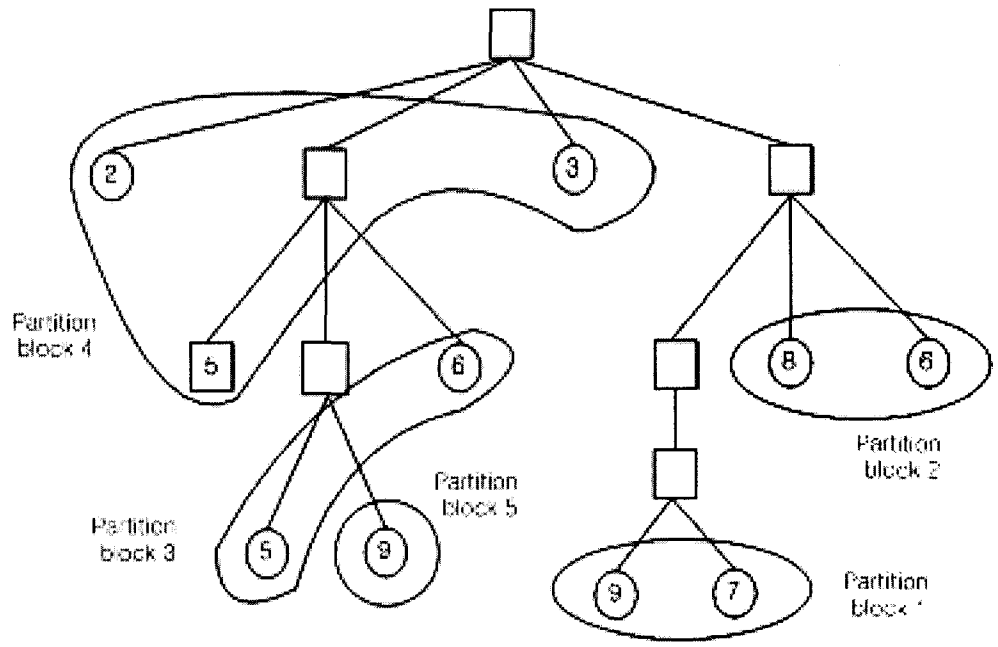
```
1:  procedure partition[] initial-partitioning(Tree cost-tree, int part-size, int l)
2:    // phase 1 : initialize c-list and p-array
3:    c-list ← all children nodes of root-node of the cost-tree
4:    p-array ← create empty partition blocks as many as part-size
5:    // phase 2 : expand, if necessary
6:    while lengthOf(c-list) < sizeOf(p-array) do
7:      if c-list contains coupled node(s) then
8:        comp ← remove a coupled node having the highest cost from c-list
9:        c-list += expand(comp)   // add all children nodes of comp to c-list
10:     else
11:       return error("can't expand...")
12:     endif
13:   endwhile
14:
15:   // phase 2.5 : expand every coupled nodes in the c-list up to l level
16:   depth = 0;
17:   while depth++ < l do
18:     t-list = null              // temporary list holding expanded nodes
19:     for each coupled model in the c-list do
20:       comp ← remove the selected coupled node from the c-llst
21:       t-list += expand(comp)
22:     endfor
23:     c-llst += t-list           // update c-list
24:   endwhile
25:
26:   // phase 3 : fill empty partition blocks in 'descending order'
27:   while an empty partition block exists in p-array
28:     comp ← remove a node having the highest cost from c-list
29:     assignTo(comp, any empty partition block in p-array)
30:   endwhile
31:   // phase 4 : distribute nodes in c-list into partition blocks in
32:                 'ascending order'
33:   while c-list is not empty do
34:     comp ← remove a node having the lowest cost from c-list
35:     assignTo(comp, the partition block having the lowest cost)
36:   endwhile
37:   return p-array
38: endprocedure
```

Algorithm 4. Initial Partitioning Algorithm in the GMP with Look-ahead(l) based on the Aggressive Look-ahead Scheme

```
1:  procedure partition[] initial-partitioning(Tree cost-tree, int part-size, int l)
2:    // phase 1 : initialize c-list and p-array
3:    c-list ← all children nodes of root-node of the cost-tree
4:    p-array ← create empty partition blocks as many as part-size
5:    // phase 2 : expand, if necessary
6:    while lengthOf(c-list) < sizeOf(p-array) do
7:      if c-list contains coupled node(s) then
8:        comp ← remove a coupled node having the highest cost from c-list
9:        c-list += expand(comp)   // add all children nodes of comp to c-list
10:     else
11:       return error("can't expand...")
12:     endif
13:   endwhile
14:   // phase 3 : fill empty partition blocks in 'descending order'
15:   while an empty partition block exists in p-array
16:     comp ← remove a node having the highest cost from c-list
17:     assignTo(comp, any empty partition block in p-array)
18:   endwhile
19:   // phase 4 : distribute nodes in c-list into partition blocks
20:    in 'ascending order'
21:   while c-list is not empty do
22:     comp ← remove a node having the lowest cost from c-list
23:     assignTo(comp, the partition block having the lowest cost)
24:   endwhile
25:
26:   // phase 4.5 : expand every coupled nodes in all partition blocks up to l times
27:   depth = 0;
28:   while depth++ < l do
29:     for each partition block in the p-array do
30:       partition block ← the selected partition block
31:       t-list = null                    // temporary list holding expanded nodes
32:       for each coupled model in the partition block do
33:         comp ← remove the selected coupled node from the partition block
34:         t-list += expand(comp)
35:       endfor
36:       partition block += t-list        // update a partition block
37:     endfor
38:   endwhile
39:   return p-array
40: endprocedure
```

Algorithm 5. Initial Partitioning Algorithm in the GMP with Look-ahead(l) based on the Deferred Look-ahead Scheme

(a) GMP with look-ahead (2) in *aggressive* scheme



(b) GMP with look-ahead (2) in *deferred* scheme

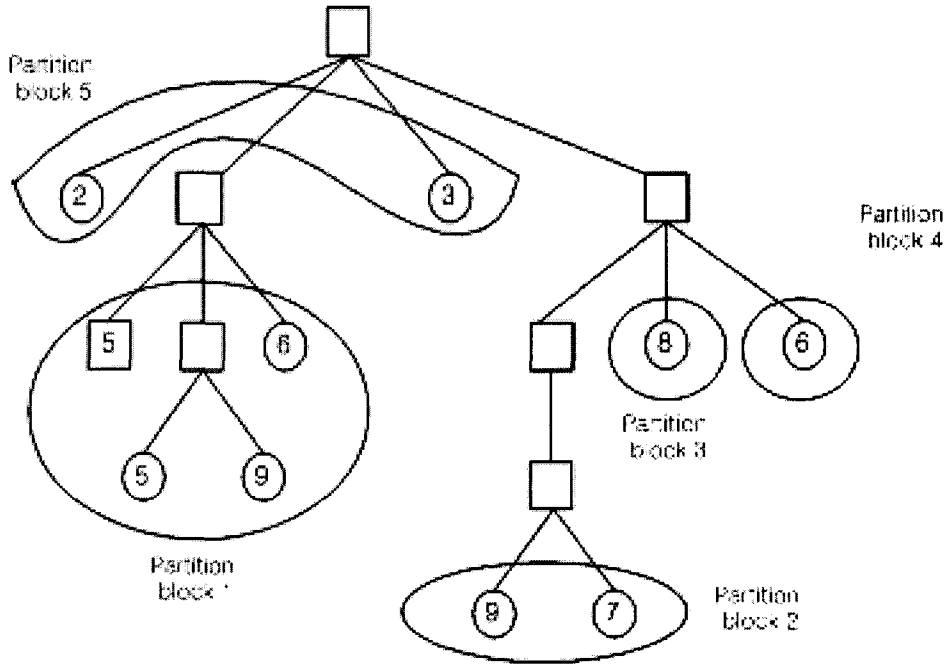


Figure 18. Initial Partitioning Results of the GMP with Look-ahead (2)

5.2.2 E^2S Partitioning with Look-ahead(l)

The E^2S partitioning in the GMP with look-ahead (*l*), (*E^2S partitioning with look-ahead(l)*), requires a second parameter *look-ahead, l*, in addition to the existing parameter; an array of partition blocks, *p-array*. E^2S partitioning algorithms of the GMP with look-ahead(*l*) for both the aggressive scheme and the deferred scheme are presented in Algorithm 6 and Algorithm 7, respectively. The modified E^2S partitioning processes and their results are illustrated in Figure 19 and Figure 20 when the look-ahead coefficient is 2 and the number of partition blocks is 5.

Figure 19.a shows the E^2S partitioning process when the aggressive scheme is used. The root node contains the result of initial partitioning, produced by the GMP-baseline algorithm ,shown in Figure 9, {{25}, {16}, {8}, {6}, {2,3}}. It is important that the root node is not strongly tied to the initial partitioning result of the proposed algorithm regardless of the look-ahead manipulation scheme applied. Thus, the node could contain an initial partitioning result produced by any GMP algorithm introduced in this paper. The partitioning result of the algorithm for n_1, which is {{9,7}, {8,6}, {6,5}, {2,3,5}, {9}}, differs from the result generated by the GMP-baseline algorithm, {{14}, {9}, {8,7}, {6,6}, {2,3,5}}. The GMP with look-ahead(2) produces a higher degree of cost homogeneity between cost nodes as compared to the GMP-baseline algorithm. This is because all coupled nodes in the partition blocks are expanded up to *l* times before assigning cost nodes in the *c-list* into those blocks again (Line 6 – Line 23 in **Algorithm 6**). Advantages of the algorithm are cost disparity reduction and partitioning tree downsizing. Similar to the initial partitioning, the cost disparity between cost nodes is

reduced in the E^2S algorithm. Also, the partitioning tree becomes smaller because the partition result rapidly approaches a best result when compared to the GMP-baseline algorithm.

Figure 19.b shows the E^2S partitioning process when the deferred scheme is used. The root node contains the initial partitioning result, produced by the GMP-baseline algorithm, shown in **Figure 9**. The partitioning result of this algorithm for n_3, is {{9,5}, {9}, {8,7}, {6,6}, {2,3,5}}, which is identical to the result generated by the GMP-baseline algorithm, which is {{14}, {9}, {8,7}, {6,6}, {2,3,5}}, in terms of cost disparity between the max- and the min-partition blocks. This is because all of the coupled nodes in the partition blocks are expanded up to *l* times after achieving a best partitioning result (Line 35 – Line 47 in Algorithm 7). This scheme creates a fine-grain partitioning result as compared to the result produced by the GMP-baseline algorithm.

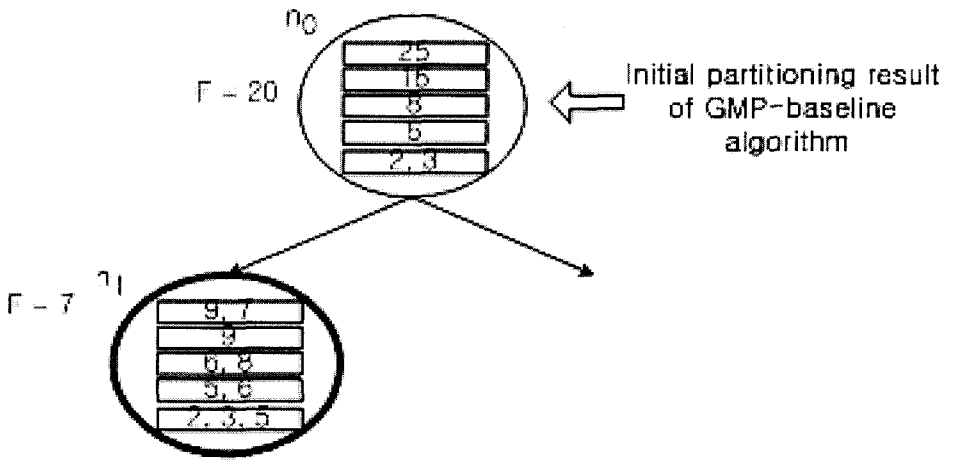
```
1:  procedure partition[] e-square-s-partitioning(partition[] p-array, int l)
2:    // phase 1: initialize e-array and e-partition
3:    e-array ← p-array
4:    c-list ← null
5:
6:    // phase 2: expand every coupled nodes in all partition blocks up to l times
7:    depth = 0;
8:    while depth++ < l do
9:      for each partition block in the e-array do
10:       partition block ← the selected partition block
11:       t-list = null                          // temporary list holding expanded nodes
12:       for each coupled model in the partition block do
13:         comp ← remove the selected coupled node from the partition block
14:         t-list += expand(comp)
15:       endfor
16:       c-list += t-list      // update a partition block
17:     endfor
18:   endwhile
19:   // phase 3: fill empty partition blocks in 'descending order'
20:   while an empty partition block exists in e-array
21:     comp ← remove the node having the highest cost in c-list
22:     assignTo(comp, any empty partition )
23:   endwhile
24:
25:   // phase 4: distribute nodes to e-array
26:   while c-list is not empty do
27:     comp ← remove a node having the lowest cost from c-list
28:     assignTo(comp, a partition of e-array having the lowest cost)
29:   endwhile
30:   // phase 5: evaluate a new partitioning result
31:   if superiorTo(evaluate(e-array), evaluate(p-array)) then
32:     return e-square-p-partitioning(e-array)
33:   else
34:     retrun p-array
35:   endif
36: endprocedure
```

Algorithm 6. E^{2}S Partitioning Algorithm in the GMP with Look-ahead(l) based on the Aggressive Look-ahead Scheme

```
1:  procedure partition[] e-square-s-partitioning(partition[] p-array, int l)
2:    // phase 1: initialize e-array and e-partition
3:    e-array ← p-array; e-node ← null
4:    e-partition ← a partition block having the highest cost in e-array
5:   // phase 2: identify an expandable partition block from e-array
6:    while true do
7:      if e-partition ≡ null then return p-array
8:      else
9:        if e-partition contains coupled node(s) then
10:         e-node ← the coupled node having the highest cost in e-partition
11:         if e-node ≠ null then break  else return p-array  endif
12:       else
13:         e-partition ← select the partition block having the highest cost from
14:                       e-array excluding previously selected partition blocks
15:       endif
16:     endif
17:   endwhile
18:   // phase 3: expand e-node and put them into c-list
19:   comp ← remove e-node from e-partition;  c-list ← expand(comp)
20:   // phase 4: fill the e-partition with the node having the highest cost
21:   // if the partition block is empty
22:   if empty(e-partition) then
23:     comp ← remove the node having the highest cost in c-list
24:     assignTo(comp, e-partition)
25:   endif
26:   // phase 5: distribute nodes to e-array
27:   while c-list is not empty do
28:     comp ← remove a node having the lowest cost from c-list
29:     assignTo(comp, a partition of e-array having the lowest cost)
30:   endwhile
31:   // phase 6: evaluate a new partitioning result
32:   if superiorTo(evaluate(e-array), evaluate(p-array)) then
33:     return e-square-p-partitioning(e-array)
34:   else do
35:   // phase 6.5 : expand every coupled nodes in all partition blocks up to l times
36:     depth = 0;
37:     while depth++ < l do
38:       for each partition block in the p-array do
39:         partition block ← the selected partition block
40:         t-list = null                          // temporary list holding expanded nodes
41:         for each coupled model in the partition block do
42:           comp ← remove the selected coupled node from the partition block
43:           t-list += expand(comp)
44:         endfor
45:         partition block += t-list   // update a partition block
46:       endfor
47:     endwhile
48:     retrun p-array
49:   endif
50: endprocedure
```

Algorithm 7. E^{2}S Partitioning Algorithm in the GMP with Look-ahead(l) based on the Deferred Look-ahead Scheme

(a) GMP with look-ahead (2) in *aggressive* scheme



(b) GMP with look-ahead (2) in *deferred* scheme

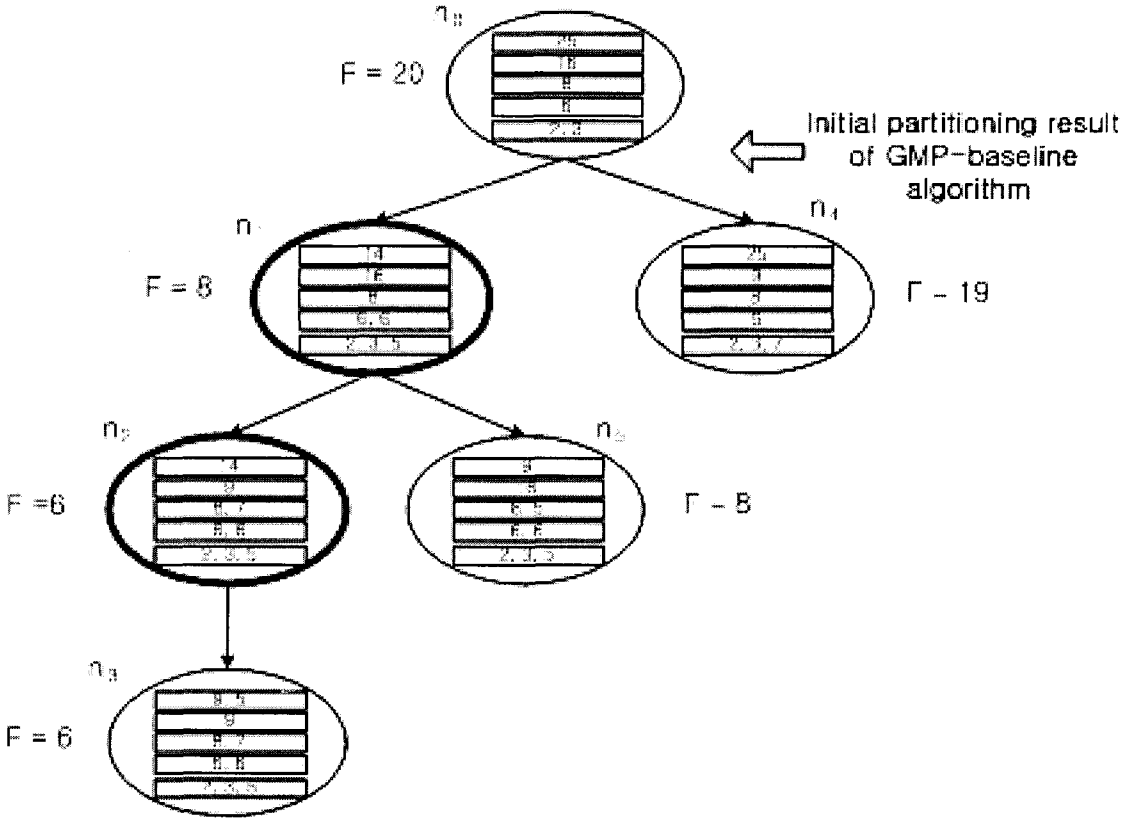
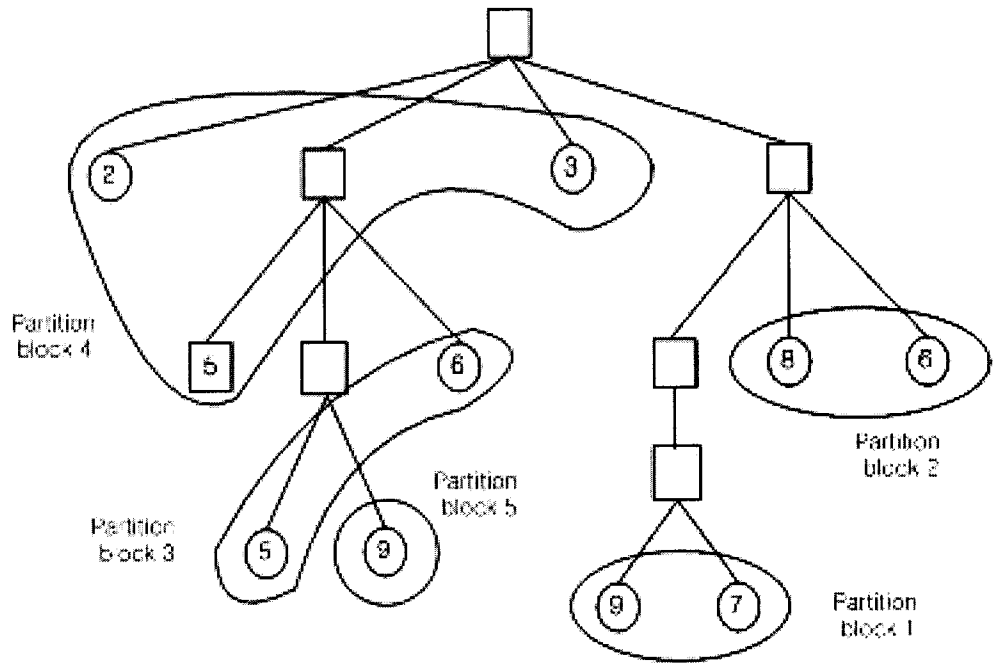


Figure 19. E^{2}S Partitioning in the GMP with Look-ahead(2)

(a) GMP with look-ahead (2) in *aggressive* scheme



(b) GMP with look-ahead (2) in *deferred* scheme

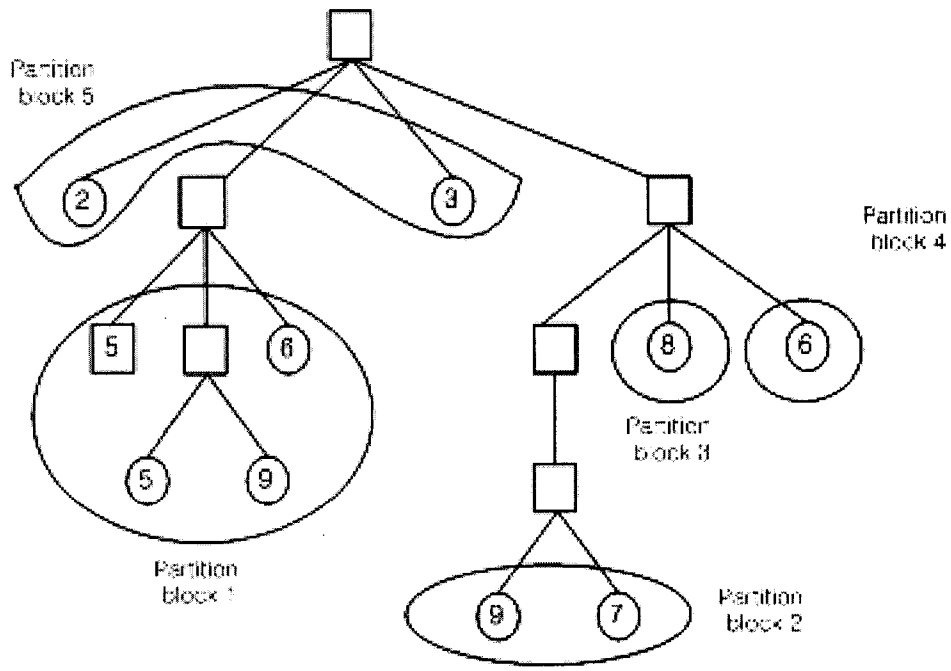


Figure 20. E^{2}S Partitioning Results of the GMP with Look-ahead (2)

5.3 Adaptive Model Partitioning (AMP) Algorithm Derived from the GMP Algorithm

An Adaptive Model Partitioning (AMP) algorithm is a GMP algorithm that partitions a hierarchical model into a set of partition blocks based on certain cost information associated with resources or hosts in a distributed system. Cost information could be either time-invariant or time-varying information about resource availability or resource capacity. Unlike previous algorithms that assume a partition block can contain unlimited number of cost nodes, the AMP algorithm presumes a partition block can enclose a certain number of cost nodes based on their aggregated cost.

The AMP algorithm conducts partitioning by assigning each cost node in the *c-list* to the partition block having the lowest cost during the partitioning process. Initial partitioning and E^2S partitioning in the AMP algorithm are presented in Algorithm 8 and Algorithm 9, respectively. Both algorithms require an additional parameter, the array of costs, *c-array*. The *c-array* serves as a decision-making criteria whenever a cost node needs to be assigned to a particular partition block. Specifically, in initial partitioning, an empty partition block is identified based on the *c-array* before filling the block with a selected node, $node_{highest}$, (Line 19 – Line 24 in **Algorithm 8**). The selected empty partition block should satisfy following requirements

$$\text{i) } p_i = empty$$
$$\text{ii) } c_i \geq node_{highest}, \quad node_{highest} \in c - list$$
$$\text{iii) } c_i - node_{highest} = \min_{j=1..P}\{c_j - node_{highest}\}$$

where, p_i : an partition block
c_i : the cost corresponded to p_i
$node_{highest}$: the cost of the node having the highest cost
P : the request number of partition blocks

Similarly, the non-empty partition block is identified before distributing remain nodes in the *c-list* to appropriate partition blocks based on the *c-array* during the distribution phase (Line 31 – Line 36 in **Algorithm 8**). The selected partition block should satisfy following requirements

i) $p_i \neq empty$
ii) $(c_i - \cos tOf(p_i)) \geq node_{lowest}, \quad node_{lowest} \in c - list$
iii) $(c_i - \cos tOf(p_i)) - node_{lowest} = \min_{j=1..P}\{(c_i - \cos tOf(p_i)) - node_{lowest}\}$

where, costOf(p_i): cost aggregation of a partition block, p_i
$node_{lowest}$: the node having the lowest cost

Algorithms for finding the partition block satisfying the above requirements are presented in **Algorithm 10** and **Algorithm 11**, respectively. In E_2S partitioning, the *c-array* is consulted when identifying an appropriate partition block before distributing remaining nodes in the *c-list* to partition blocks (Line 40 – Line 45 in Algorithm 9). Even though resource capacity conceptually differs from resource availability, the adaptive algorithm treats them identically.

The following sub-chapters describes how GMP algorithms for various distributed systems (i.e., GMP-SHM, GMP-SHT, GMP-DHM, and GMP-DHT) can be implemented using the AMP algorithm with appropriate cost information as well as legacy information (i.e., a cost tree and the number of partition blocks).

```
1:  procedure partition[]
2:     initial-partitioning(Tree cost-tree, int part-size, cost[] c-array)
3:     // phase 1 : initialize c-list and p-array
4:     c-list ← all children nodes of root-node of the cost-tree
5:     p-array ← create empty partition blocks as many as part-size
6:     // phase 2 : expand, if necessary
7:    while lengthOf(c-list) < sizeOf(p-array) do
8:       if c-list contains coupled node(s) then
9:          comp ← remove a coupled node having the highest cost from c-list
10:         c-list += expand(comp)   // add all children nodes of comp to c-list
11:      else
12:         return error("can't expand...")
13:      endif
14:   endwhile
15:
16:   // phase 3 : fill empty partition blocks in 'descending order'
17:   while an empty partition block exists in p-array
18:      comp ← remove a node having the highest cost from c-list
19:      pi ← find-best-emptyblock (p-array, c-array, costOf(comp))
20:      if(pi != null) then
21:         assignTo(comp, pi)
22:      else
23:         return error("can't assign...")
24:      endif
25:    endwhile
26:
27:   // phase 4 : distribute nodes in c-list into partition blocks in
28:                'ascending order'
29:   while c-list is not empty do
30:     comp ← remove a node having the lowest cost from c-list
31:     pi ← find-best-block (p-array, c-array, costOf(comp))
32:     if(pi != null) then
33:        assignTo(comp, pi)
34:     else
35:      return error("can't assign...")
36:     endif
37:   endwhile
38:  return p-array
39: endprocedure
```

Algorithm 8. Adaptive Model Partitioning (AMP) Algorithm for Initial Partitioning

```
1:  procedure partition[] e-square-s-partitioning(partition[] p-array, cost[] c-array)
2:    // phase 1: initialize e-array and e-partition
3:    e-array ← p-array
4:    e-partition ← a partition block having the highest cost in e-array
5:    e-node ← null
6:    // phase 2: identify an expandable partition block from e-array
7:    while true do
8:      if e-partition ≡ null then return p-array
9:      else
10:       if e-partition contains coupled node(s) then
11:         e-node ← the coupled node having the highest cost in e-partition
12:         if e-node ≠ null then break  else return p-array  endif
13:       else
14:         e-partition ← select the partition block having the highest cost from
15:                       e-array excluding previously selected partition blocks
16:       endif
17:     endif
18:   endwhile
19:   // phase 3: expand e-node and put them into c-list
20:   comp ← remove e-node from e-partition
21:   c-list ← expand(comp)
22:
23:   // phase 4: fill the e-partition with the node having the highest cost
24:   // if the partition block is empty
25:   if empty(e-partition) then
26:     comp ← remove the node having the highest cost in c-list
27:     assignTo(comp, e-partition)
28:   endif
29:   // phase 5: distribute nodes to e-array
30:   while c-list is not empty do
31:     comp ← remove a node having the lowest cost from c-list
40:     find-best-block (p-array, c-array, costOf(comp)), pi)
41:     if(pi != null) then
42:       assignTo(comp, pi)
43:     else
44:       return error("can't assign...")
45:     endif
32:   endwhile
33:   // phase 6: evaluate a new partitioning result
34:   if superiorTo(evaluate(e-array), evaluate(p-array)) then
35:     return e-square-p-partitioning(e-array)
36:   else
37:     retrun p-array
38:   endif
39: endprocedure
```

Algorithm 9. Adaptive Model Partitioning (AMP) Algorithm for E^2S Partitioning

```
1:  procedure partition
2:    find-best-emptyblock(partition[] p-array, cost[] c-array, cost comp)
3:      collection ← all empty partition blocks in the p-array
4:      disparity ← INFINITY    // cost disparity
5:      best-fit = NULL
6:      foreach a partition block, p_i, in the collection do
7:        c_i = the total cost associated with p_i  // c_i = c-array[i] and p_i = p-array[i]
8:        if  (c_i ≥ comp) and (c_i – comp) ≤ disparity then
9:          disparity = c_i – comp
10:         best-fit = p_i
11:       endif
12:     endforeach
13:     return best-fit
14: endprocedure
```

Algorithm 10. Algorithm for Finding an Empty Partition Block Having Minimum Cost Disparity Compared to a Given Cost

```
1:  procedure partition
2:    find-best-block(partition[] p-array, cost[] c-array, cost comp)
3:      collection ← all partition blocks in the p-array
4:      disparity ← INFINITY    // cost disparity
5:      best-fit = NULL
6:      foreach a partition block, p_i, in the collection do
7:        c_i = the total cost associated with p_i  // c_i = c-array[i] and p_i = p-array[i]
8:        limit = c_i – costOf(p_i);
9:        if  (limit ≥ comp) and (limit – comp) ≤  disparity then
10:         disparity = limit – comp
11:         best-fit = p_i
12:       endif
13:     endforeach
14:     return best-fit
15: endprocedure
```

Algorithm 11. Algorithm for Finding a Partition Block Having Minimum Cost Disparity Compared to a Given Cost

5.3.1 GMP Algorithm for Static Information/Heterogeneous System (GMP-SHT)

A generic *model partitioning algorithm for static information/heterogeneous system*, GMP-SHT, is implemented by providing stationary host information, *t-array*, along with legacy information into the AMP algorithm. The *t-array* represents total costs of hosts in a heterogeneous system. In the algorithm, a partition block is a hormomorpic representation (or proxy) of the host. Thus, the aggregated cost of a partition block cannot exceed the total cost of its corresponding host. Similar to the GMP-SHM, the GMP-SHT algorithm produces an identical partitioning result regardless of how often it is applied to a given model. By applying the GMP-DHM algorithm to a distributed system that is comprised of homogeneous hosts, we can build a static resource allocator or distributor.

5.3.2 GMP Algorithm for Dynamic Information/Homogeneous System (GMP-DHM)

A generic *model partitioning algorithm for dynamic information/homogeneous system*, GMP-DHM, is implemented by providing time-varying resource information, *a-array*, along with legacy information to the AMP algorithm. The *a-array* represents time-varying resource availability of hosts, *available costs*, in a homogeneous system. Similar to the GMP-SHT, the aggregated cost of a partition block cannot exceed the available cost of its corresponding host. Since the GMP-DHM algorithm performs partitioning based on time-varying information, the algorithm may produce a different partitioning result when it is executed again for the given model. Monitoring and obtaining available costs are not represented in the algorithm. The algorithm does not describe the ways of monitoring or obtaining time variant cost information. Instead, The algorithm presumes

that available cost information is supplied from outside. By integrating the GMP-DHM algorithm with a system monitor that observes activities of hosts in the homogeneous system, we can build a dynamic load distributor.

5.3.3 GMP Algorithm for Dynamic Information/Heterogeneous System (GMP-DHT)

A generic *model partitioning algorithm for dynamic information/heterogeneous system*, GMP-DHT, is implemented by providing time-varying host information, *t-array*, along with legacy information to the AMP algorithm. The *t-array* represents time-varying resource capacity of hosts, *total costs*, in a heterogeneous system. Similar to the GMP-DHM algorithm, the GMP-DHT algorithm may produce a different partitioning result when it is executed again for the given model. By integrating the GMP-DHT algorithm with a resource monitor that observes time-varying resource allocation status in the heterogeneous system, we can build an adaptive resource allocator/optimizer.

CHAPTER 6. APPLICATIONS

This chapter introduces a few applications that demonstrate the usability and adaptability of the GMP algorithms in real world problems. For distributed simulation, the DEVS Modeling and Simulation (M&S) framework over advanced distributed network systems (DEVS/ADNS) is introduced. Also, resource allocators are presented to show how the GMP algorithms are applied to various classical distributed computing problems. Finally, the N-body mapping problem is discussed to show the usability of the GMP algorithms in scientific computing applications.

6.1 DEVS M&S Framework over the Advanced Distributed Network System (DEVS/ADNS)

Several DEVS M&S frameworks have been developed to meet numerous M&S demands on various distributed network systems. DEVS-CORBA and DEVS-HLA are some of them. DEVS-CORBA is a DEVS M&S framework that runs on industry-standard distributed network middleware, CORBA. DEVS-CORBA serves as a solid M&S solution for large-scale M&S problems involving sophisticated distributed systems that are comprised of heterogeneous system elements such as OS, computer language, and network technology.

DEVS-HLA is a DEVS M&S framework that runs on RTI environment. The RTI is a HLA implementation over a TCP/IP network. DEVS-HLA is a HLA-compliant simulation framework and is the foundation of Joint MEASURE™ M&S environment at Lockheed-martin. Both DEVS-CORBA and DEVS-HLA allow users to perform

modeling and simulation without any knowledge of the underlying network middleware (i.e., CORBA and RTI). DEVS/ADNS is a new DEVS M&S framework targeting emerging distributed network systems such as Peer-to-Peer (P2P) and GRID with support for hierarchical model partitioning, autonomous model deployment, self-system configuration, and coupling restructuring. The DEVS/ADNS supports both P2P and GRID at this time. Legacy environments such as CORBA and HLA could be supported by implementing DEVS/ADNS wrappers. GMP algorithms are applied to implement key components of the DEVS/ADNS; the model partitioner and model deployer.

DEVS/ADNS has five layers; network infrastructure, network middleware layer, simulation layer, modeling layer, and application layer from the bottom to the top as shown in **Figure 21**. The *network infrastructure layer* describes various network infrastructures. P2P, GRID, CORBA, and HLA are some examples. The P2P is an emerging distributed network infrastructure in the Internet community. Unlike other existing infrastructures, it finds other peers (or hosts) and shares information without centralized servers using certain peer discovery protocols and information sharing protocols in a loosely coupled network system such as the Internet. The GRID is a new distributed network infrastructure interconnecting high performance computers (e.g., super computers), instruments, and storage devices through (ultra) high-speed networks[45].

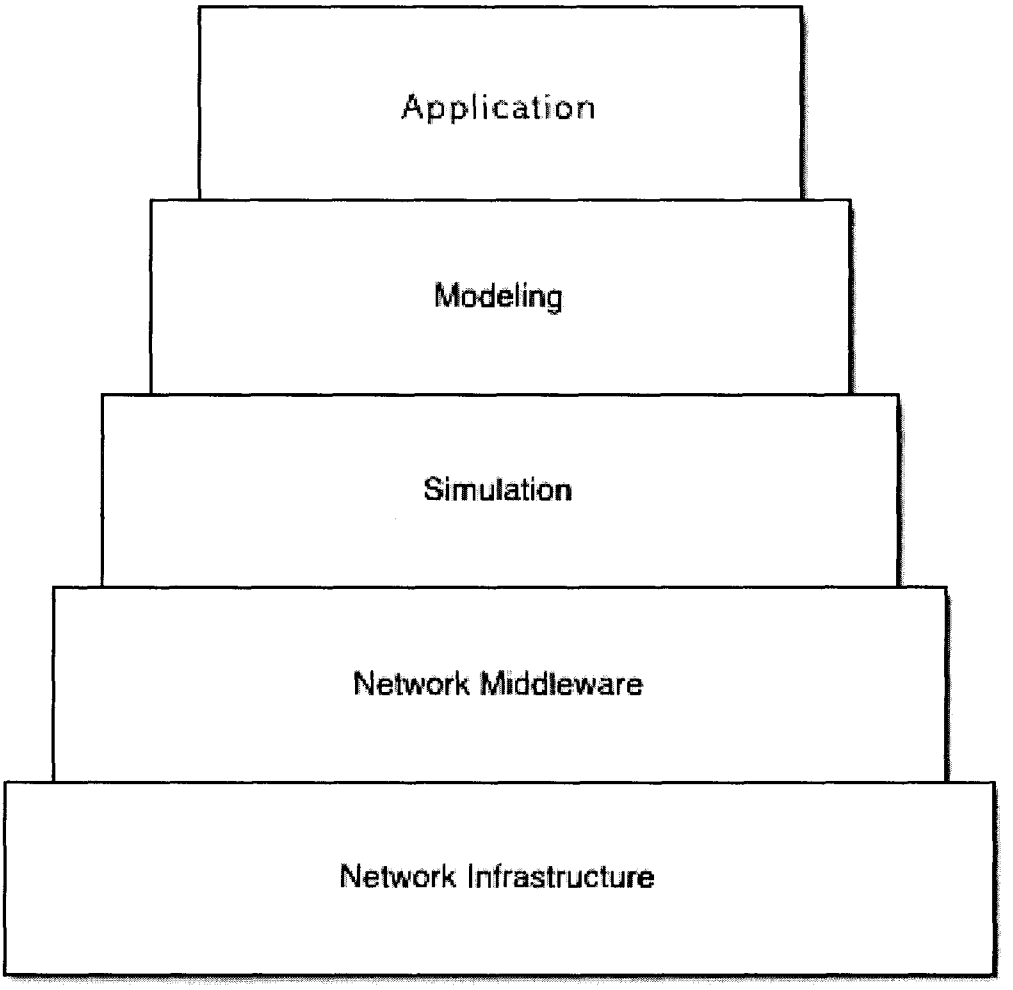


Figure 21. Layered Architecture of DEVS/ADNS

It is primarily targeted at large-scale and challenging problems in the scientific and engineering domain (e.g., the search for the origin of galaxy) rather than general purpose distributed computing problems. Thus, efficient resource management is the one of the key topics in GRID community. CORBA is an industry-standard distributed network infrastructure for general purpose distributed computing with support for language-independent and platform-independent capabilities[25]. With successful development and deployment, CORBA has become a major network middleware in many application domains (e.g., telecommunication). The HLA is a DoD-initiated distributed network infrastructure mainly targeted at distributed simulation. The goal of the HLA is to achieve a high degree of reusability of simulation models and interoperability between heterogeneous simulation components, rather than high performance distributed simulation. Current implementation of DEVS/ADNS runs on both P2P network and GRID. Various distributed network infrastructures could be easily embraced by the DEVS/ANDS.

The *network middleware layer* provides a set of standard network interfaces that are independent of the underline network infrastructure. The interfaces could be implemented by using existing network middleware or creating new middleware from scratch. In its current implementation, DEVS/ADNS adopts JXTA for P2P and Globus for GRID[46, 47]. It plans to support RTI for HLA and Visibroker™ for CORBA. The JXTA is a joint effort of Internet community aimed at building and deploying XML-based P2P network systems for all network devices. The Globus is one of GRID implementations. The RTI is a runtime environment for HLA simulation components and is also a reference

implementation of HLA. The Visibroker is the most famous CORBA implementation for TCP/IP network infrastructures. It provides the major features specified by the OMG CORBA standard along with some additional capabilities such as smart agents and an object activation daemon.

The *simulation layer* provides a set of components and services for distributed simulation in the DEVS/ADNS framework. Those services includes the *DEVS Naming and Directory service (DEVS N&D)* which allows identification of modeling and simulation components without any knowledge of the underline network infrastructure, the *hierarchical model partitioning* which divides a hierarchical modular DEVS model into a set of partition blocks, *autonomous model deployment* which dispatches a set of partition blocks to simulators that are dispersed over a distributed network, a *self-configuration* service that launches simulators with appropriate DEVS models and creates and interconnects communication channels before starting simulation cycles, etc. Those services are realized by cooperation between the model partitioner, model deployer, activators, and simulators as shown in **Figure 22**. The model partitioner is implemented by the GMP-baseline algorithm. For the given DEVS model and the requested number of partition blocks, the partitioner builds a set of partition blocks so that each block contains at least one DEVS model. To achieve better partitioning results, it may be possible to use other GMP algorithms such as GMP with look-ahead(l) and GMP-DHM instead of the GMP-baseline. The deployer dispatches a partition block with appropriate coupling information into an activator that exists on a host. Upon receiving a partition block, the activator launches a simulator for each model in the block.

The *modeling layer* allows users to build simulation models without any knowledge of the underline network infrastructure. Only domain knowledge of DEVS modeling is required. That is the one of main benefits of layered design. The *application layer* is related to actual M&S problems such as supply chain management and scientific computation.

The GMP-baseline algorithm is used to realize the model partitioner in DEVS/ADNS. For a given DEVS model, a cost tree is constructed from the model by parsing the model and extracting cost information. Various cost measures could be used to capture cost information of the model. The number of internal states, the number of ports, and activity are some of them. Upon obtaining cost information, the partitioner conducts partitioning based on the proposed GMP-baseline algorithm and produces a set of partition blocks in which each block has at least one component model. Using the original coupling information of the given model and the newly created partition blocks, the coupling information is restructured. However, only couplings corresponded to components that are expanded and partitioned are restructured. Other couplings remain intact. **Figure 23** shows a model partitioner and its result, *payload*. The payload is an array of pairs in which a pair contains a set of models and their coupling information. The model deployer dispatches the payload to a host. **Figure 24** illustrates an example of coupling restructuring when the requested number of partition blocks is 3.

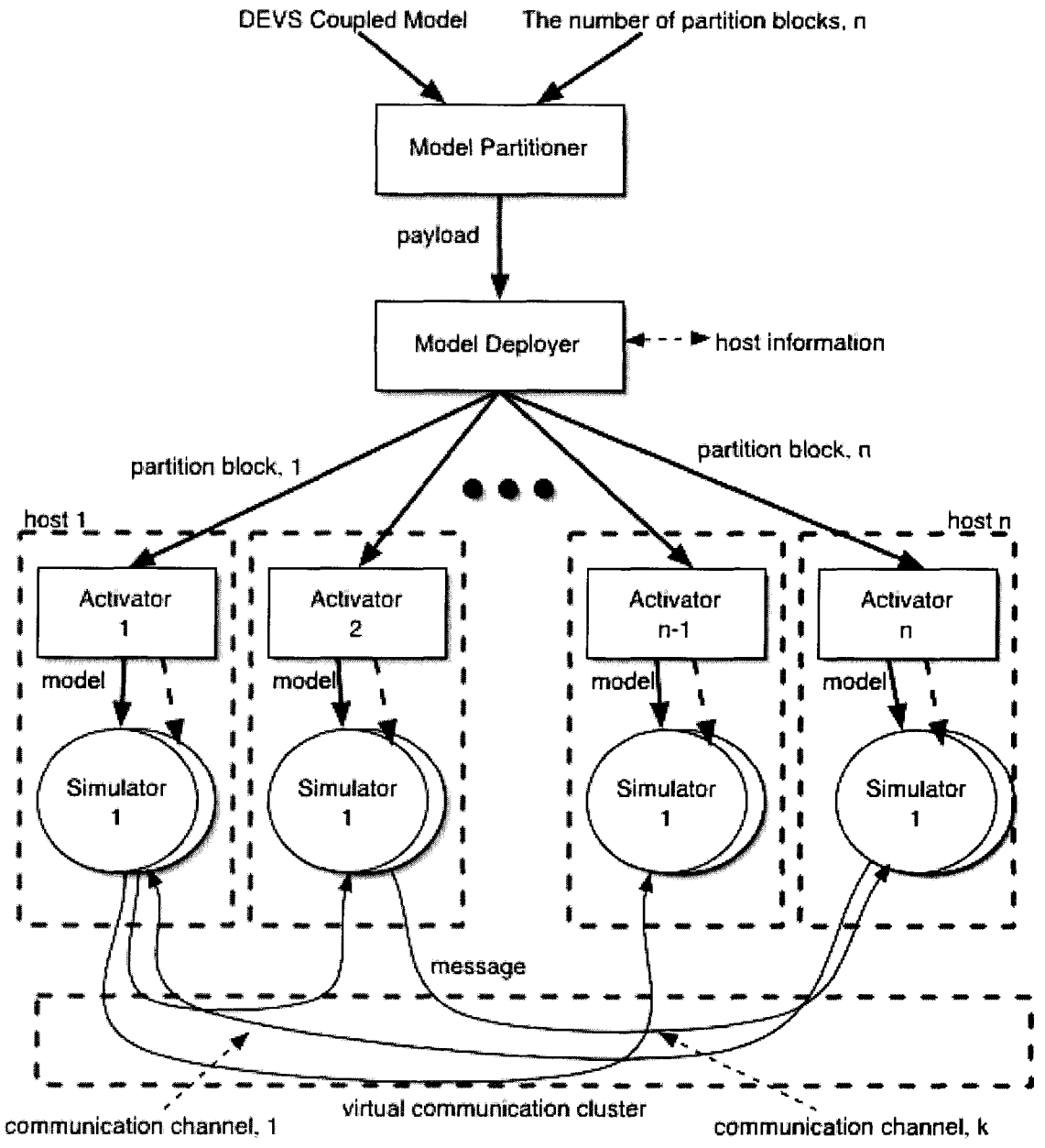
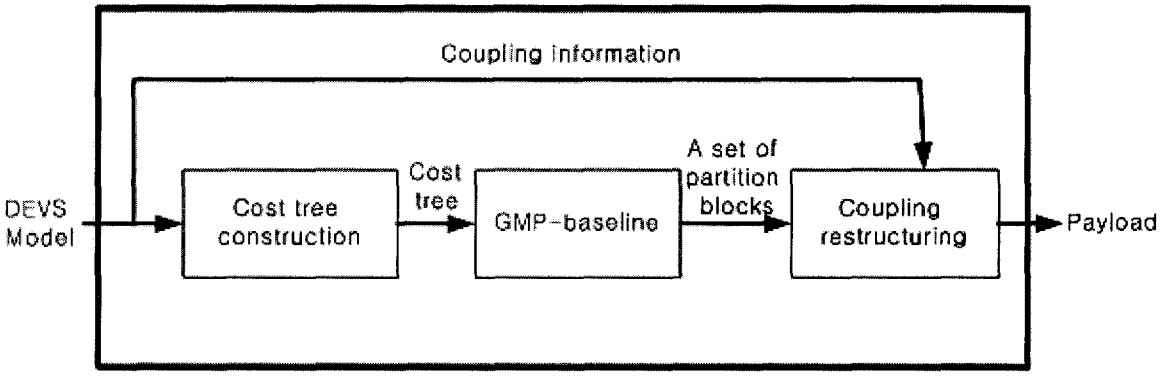


Figure 22. DEVS/ADNS Components: Partitioner, Deployer, Activator, and Simulator

Simulation is initiated by launching a simulator for a model. Once launched, each simulator initializes its simulation environment, sets up communication channels with other simulators, and waits until all communication channels are established. Information on other simulators is specified in the coupling information. After finishing communication channel setup, the actual simulation cycle is performed based on the DEVS P2P protocol. The protocol realizes a DEVS simulation without coordinators by using the publish/subscribe paradigm along with two synchronization points (i.e., barriers) as shown in **Figure 25**. For more information on DEVS/ADNS over P2P network, see [29].

(a) Model partitioner



(b) Payload

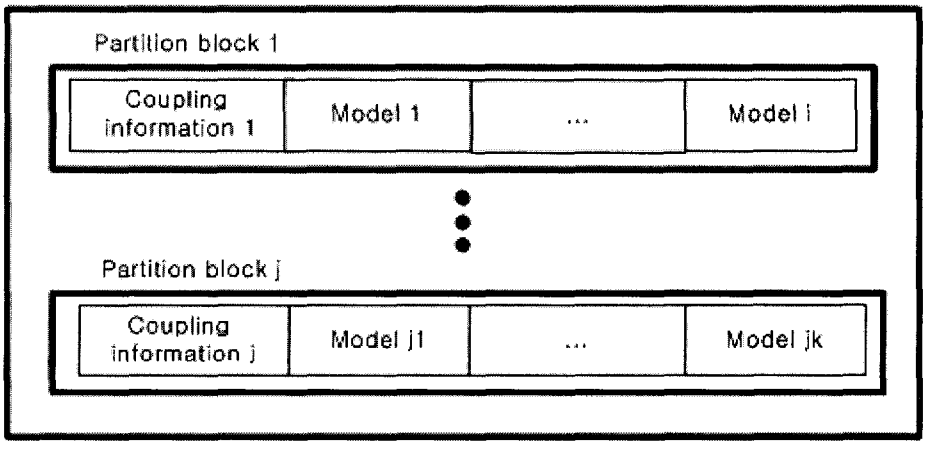
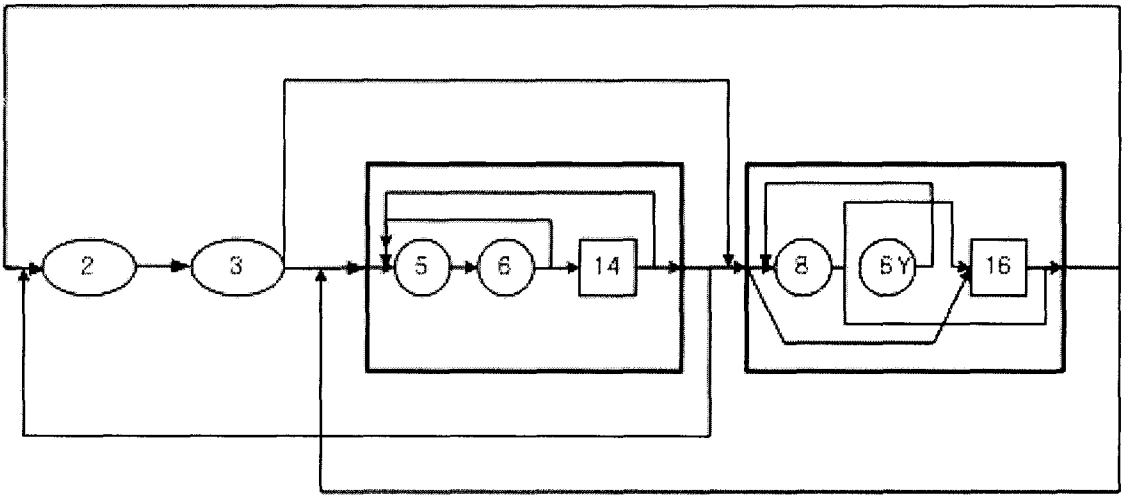


Figure 23. Model Partitioner for the DEVS/ADNS

(a) Original coupling information



(b) Restructured coupling information when the number of partition block is 3

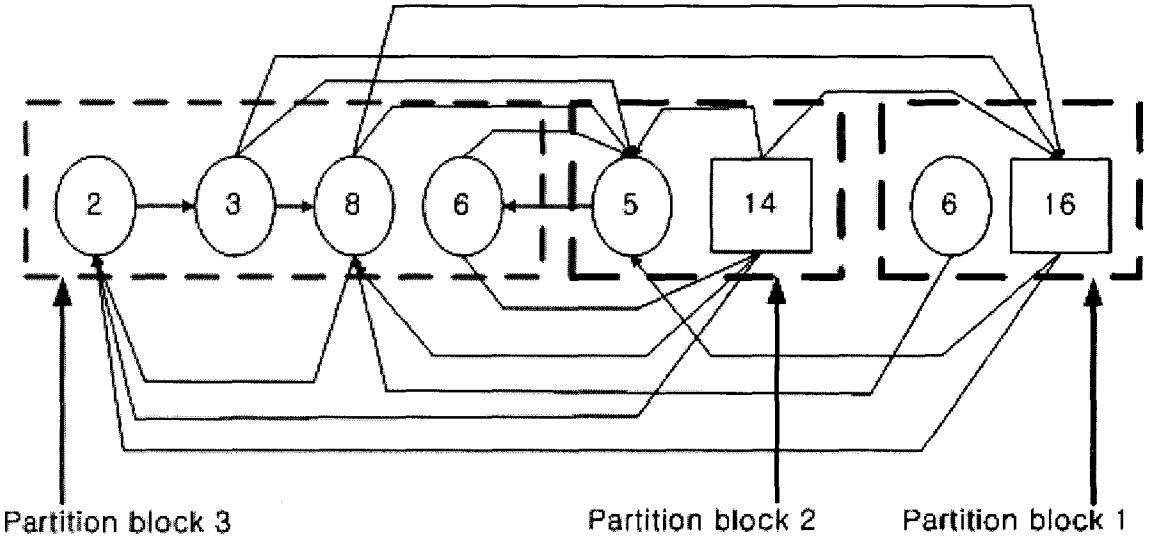


Figure 24. An Example of Coupling Restructuring

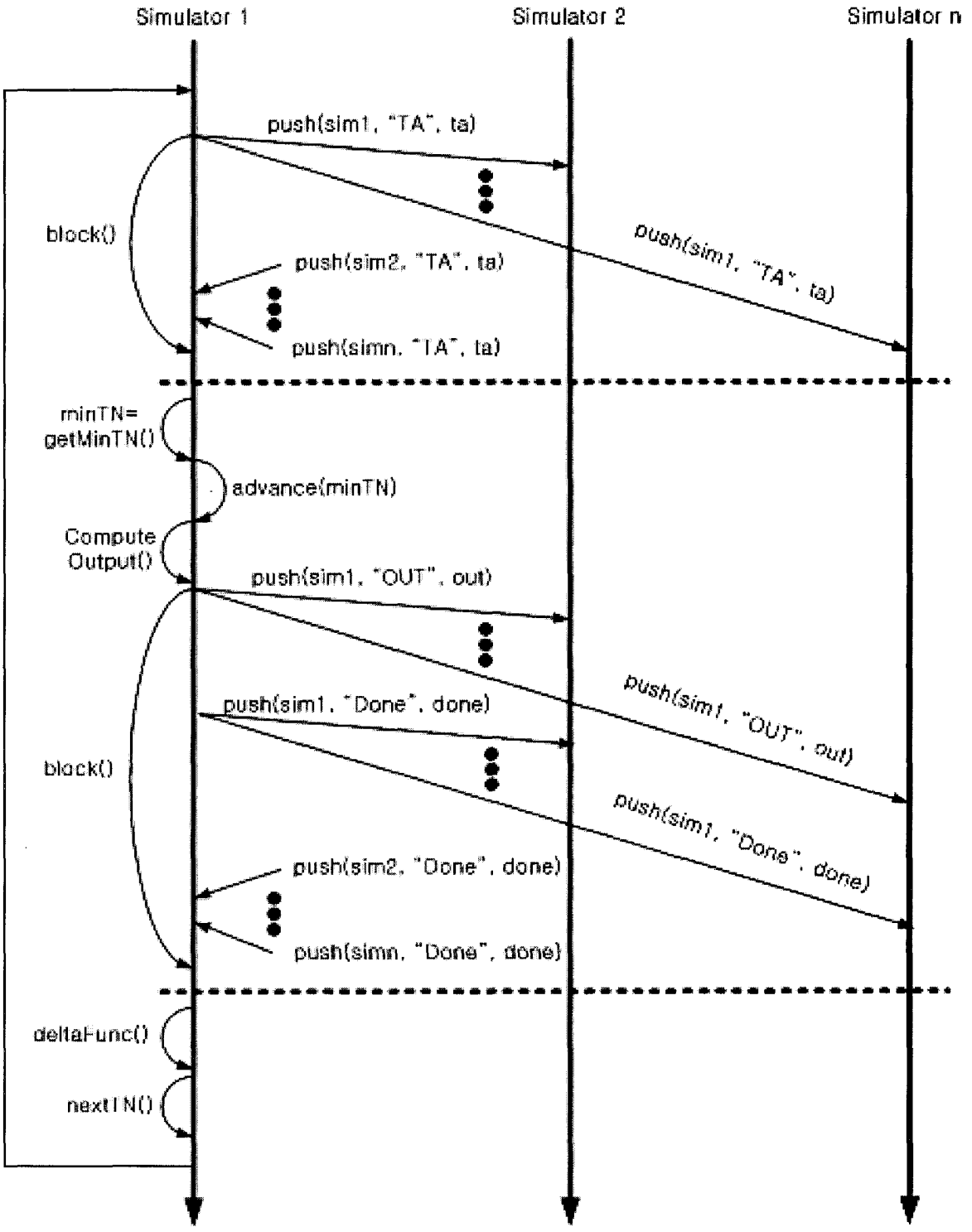


Figure 25. DEVS P2P Simulation Protocol

6.2 Resource Allocator

Resource allocation is one of the most important tasks in a distributed system. Efficient resource allocation and management is crucial to maintaining system health. The GMP-SHT algorithm provides a basis for implementing an advanced static resource allocator that evaluates sophisticated resource allocation request and sends partitioned requests to appropriate resource providers. The resource allocation request is characterized by a resource allocation model. The model is a hierarchical, modular tree structure in which an indecomposable node contains the resource allocation request for a single resource and a decomposable node is an aggregation of various resource allocation requests. A resource provider is a host or any active system component that can supply requested resources.

The static allocator based on the GMP-SHM algorithm has two components; *resource (allocation) parser* and *GMP-SHM partitioner*. The resource allocation parser creates a cost tree by evaluating an incoming resource allocation model with certain cost measures. Upon receiving the cost tree, the GMP-SHM partitioner performs partitioning and produces a set of partition blocks. Resource allocation is completed by dispatching those blocks into resource producers. The allocator may not guarantee optimality of resource allocation because resources are allocated to a set of resource providers without any consideration of their time-varying resource allocation information. It is the fundamental limitation of resource allocation based on static information.

An adaptive resource allocator based on the GMP-DHT algorithm dispatches a set of resource requests to resource providers with consideration for their time-varying resource

allocation status. The allocator has three components; *resource allocation parser, GMP-DHT partitioner,* and *resource monitor*. The resource parser acts the same as the parser of the static allocator. The GMP-DHT partitioner performs partitioning based on the GMP-DHT algorithm to create a set of partition blocks, and the resource monitor observes time-varying resource allocation status of resource providers. Upon receiving a cost tree from the parser, the GMP-DHT partitioner retrieves time-varying resource allocation information from the resource monitor and performs partitioning using the GMP-DHT algorithm. A certain level of resource allocation optimality is achieved by the adaptive resource allocator. By considering time-varying resource status of resource providers, the allocator creates partition blocks that are customized to their resource availability or resource capacity. This allows each provider to supply requested resources more efficiently. Even though a particular number of partition blocks is not provided, the allocator can find the optimal number of partition blocks and allocates resources into appropriate providers by reflecting time-varying resource information available to the allocator.

The proposed resource allocators can emulate a conventional resource allocator, which dispatches a series of non-hierarchical resource allocation requests to resource providers, by "serializing" a hierarchical resource allocation model into an array of non-hierarchical resource allocation requests. This is easily achieved by setting the number of partition blocks to the total number of decomposable nodes in the model. Owing to the heterogeneous nature of the GMP algorithms, allocators based on these algorithms could be used as resource schedulers, too.

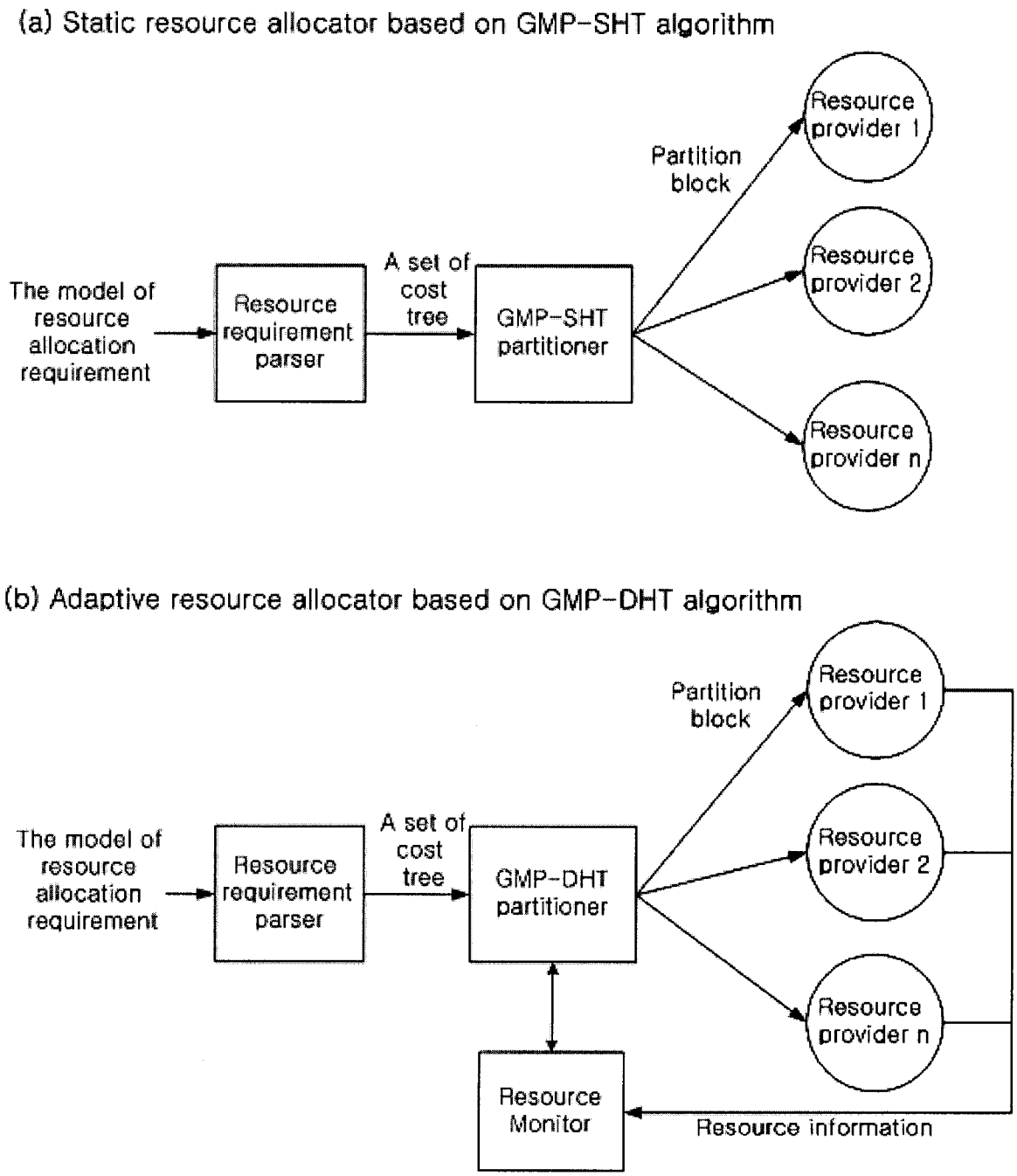


Figure 26. Resource Allocators based on GMP Algorithms

6.3 N-body Mapping Problem

The N-body problem is a famous scientific computing problem that is widely applied in many areas of science and engineering. It simulates the evolution of a galaxy that consists of n bodies (or stars). In the problem, stars move in space based on their speed and gravity. Numerous studies have been conducted to identify stars' coordinates at particular time correctly and rapidly. The Barnes-Hut algorithm is an N-body solving algorithm[48]. In the Barnes-Hut algorithm, spatial information for stars is represented by a hierarchical tree. After determining stars' new coordinates from the previous ones, the tree is reconstructed to reflect the latest spatial position of the stars. This is repeated until the algorithm terminates.

GMP algorithms could be used as a part of an N-body solver to reduce the time spent in computing stars' coordinates by allocating stars to hosts (or, processors) efficiently and executing them in parallel. Figure 27 shows how stars are mapped from space to a set of hosts using the GMP-SHM algorithm.

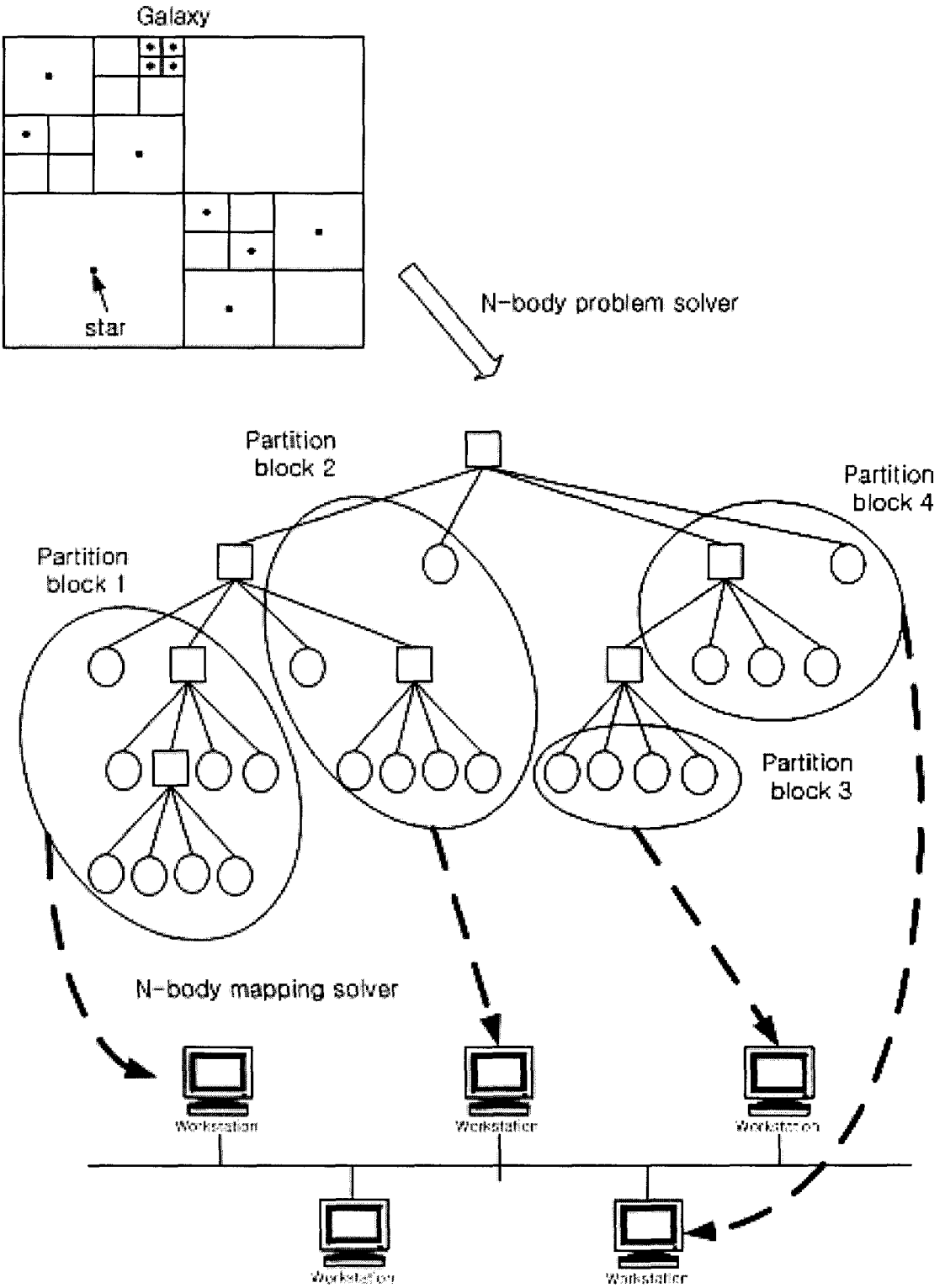


Figure 27. N-body Mapping Solver based on GMP-Baseline Algorithm

CHAPTER 7. RELATED RESEARCH

In this chapter, two major approaches to partitioning hierarchical DEVS models are introduced and compared to the GMP algorithms. The first approach is the HIPART algorithm proposed by Kim, etc.[49] and the other is the ENCLOSE algorithm proposed by Zhang, etc.[50] These algorithms are briefly described, with an example, and compared to GMP algorithms in various aspects.

7.1 HIPART: Kim's Approach

The HIPART algorithm is a partitioning algorithm for DEVS models aimed at minimization of simulation execution time in an optimistic distributed simulation environment. To achieve this goal, this algorithm conducts partitioning so as to equalize computational loads between partition blocks and to minimize communication overhead between simulators. In the algorithm, a hierarchal DEVS model is divided into a set of partition blocks based on computation and communication costs of the model. Cost information of the model is represented by a task tree. The task tree describes computation cost of each component model as a node and communication cost between two components as a direct link between them. Computational cost of a node is specified by a pair of two costs; the first represents the computational cost of the node and the second is the aggregated cost of its children and the node itself. For example, a coupled model, which has its own computation cost, 10, and the aggregated costs of its children, 200, is represented by <10, 210> in a task tree. An atomic model having computational cost, 100, is represented by <100, 100> as shown in Figure 28.a.

The partitioning process is initiated by computing the average computation cost of the given model, L*avg*. The average is easily calculated by dividing the cost of the model by the total number of partition blocks. Once L*avg* is determined, nodes with computational cost greater than L*avg* are chosen from child nodes of the root node in the task tree. Among them, the node having the largest cost disparity as compared to L*avg* and its child nodes is selected. The node having the minimum cost disparity compared to L*avg* in these nodes is identified as a partition block. After assigning the subtree starting from the identified node to a partition block, computational costs of all parent nodes of the node and L*avg* is recomputed without the node. This is repeated until no more nodes greater than L*avg* exist in the task tree. Thereafter, the node having minimum cost disparity compared to L*avg* is assigned to a partition block until every partition block is filled with a node (or subtree). Figure 28.b shows a partitioning result of the HIPART algorithm for the task tree shown in Figure 28.a where the total number of partition blocks is 3.

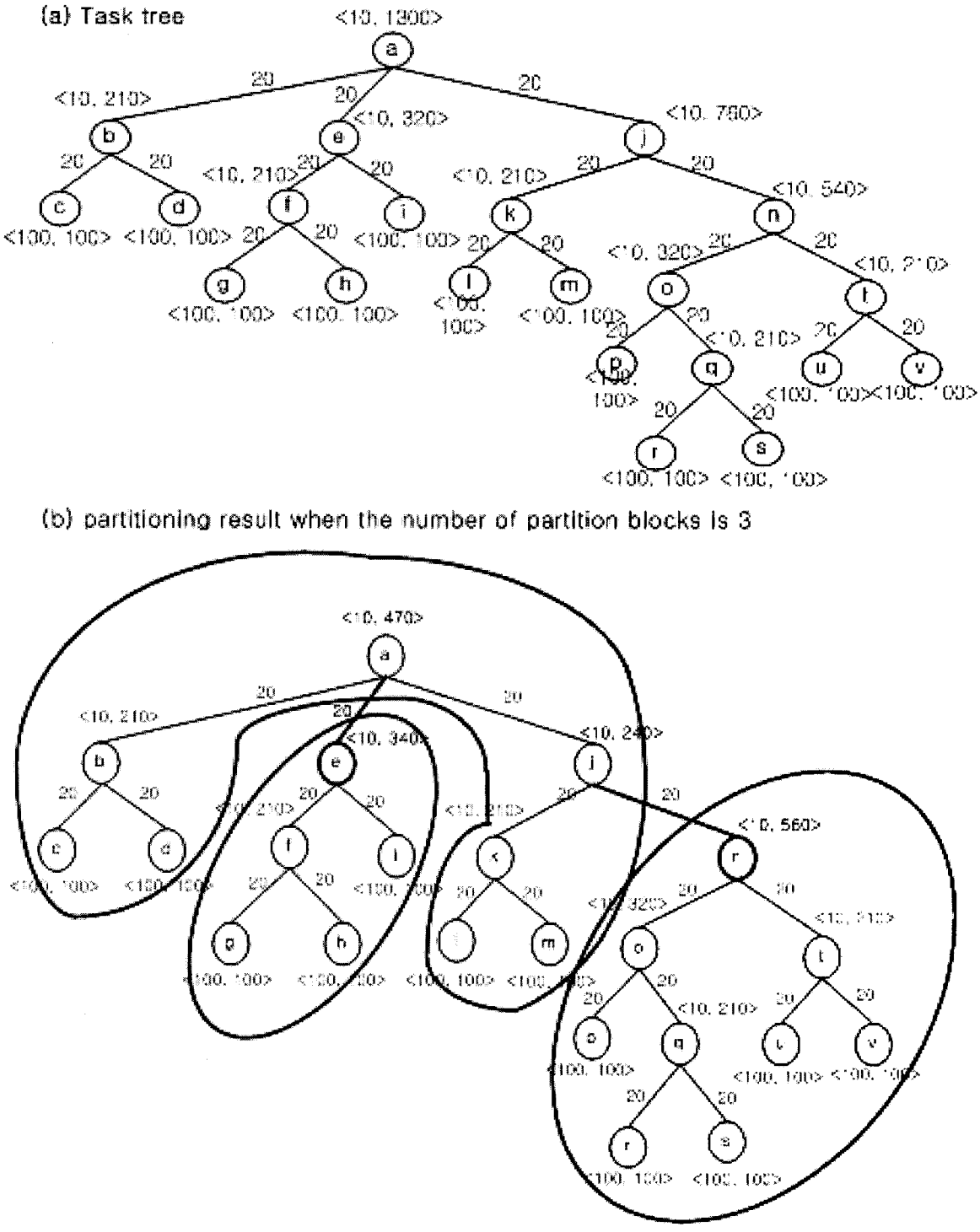


Figure 28. A Task Tree and Partitioning Result in Kim's Approach [49]

7.2 ENCLOSE: Zhang's Approach

The ENCLOSE algorithm is a partitioning algorithm for DEVS models that divides a given model into a set of partition blocks based on simulation execution time analysis, and after identifying the optimal number of partition blocks for achieving the smallest simulation execution time. The simulation execution time is obtained by running the given model on a single processor and measuring the execution time. To collect optimal execution time information on a single processor, the simulation is performed using models that are fully decomposed from the given model. The execution time information includes communication time between simulation components (e.g., coordinator and simulator), table lookup time taken by a coordinator, time spent for identifying imminent models, time spent for handling external and internal events, and time required to compute internal transition functions and generating output messages.

The partitioning process is initiated by assigning the number of partition blocks to zero and checking the type of the given model. When the model is a root node, the number is increased to 1. If the model is atomic, the algorithm terminates and returns the optimal processing time of the model. Otherwise, execution time between sequential execution and distributed execution regarding the model is compared. If sequential execution time is equal to or less than distributed execution time, the algorithm assigns the model and its child nodes to a single partition and then terminates. If not, for each child of the model, the number of partition blocks is increased by 1 and each child node is assigned to a partition block.

7.3 Comparison to GMP Algorithms

GMP algorithms are much more flexible and generic as compared to the above approaches in terms of generating, evaluating, and maintaining cost information required by the partitioning process. The approaches introduced above rely on runtime information for performing partitioning; a task tree for HIPART and simulation execution time table for ENCLOSE. In Kim's approach, a methodology of collecting or generating computational cost and communication cost for the given model is not clearly described. Generally, computational cost and communication cost are obtained by running the model for a specific amount of time or collecting necessary information during simulation. Similarly, in Zhang's approach, the execution time of the given model is obtained by running the model on single processor for a certain amount of time. Flattening the model and simulation of flattened models are time-consuming operations.

The proposed GMP algorithms do not depend on runtime information. With the heterogeneous nature of cost analysis, either static or runtime information is accepted by the GMP algorithms. Cost information is directly extracted from the given model by applying a certain cost measure. If necessary, a dynamic cost measure, such as activity, is applied to the model to emulate the above approaches for building partition blocks that produce the smallest simulation execution time.

The GMP algorithms can find the optimal number of partition blocks for the given model without any runtime information. In Kim's approach, the number of partition blocks should be provided before starting the simulation. In Zhang's approach, the optimal number of partition blocks is identified by consulting runtime information (i.e.,

simulation execution time). The information should be obtained by running the model on a single processor before performing partitioning. In the GMP algorithms, the optimal number of partition blocks can be directly identified from the given model without any runtime information, as described in chapter 4.

The GMP algorithms provide various partitioning granularity. Both HIPART and ENCLOSE produce the partitioning result without consideration of partitioning granularity. The GMP algorithms can produce a partitioning result having appropriate partitioning granularity for the various distributed systems that are involved in the partitioning. This makes the GMP algorithms adaptable to network topology and system configuration.

The GMP algorithm supports incremental quality of partitioning. In the above approaches, partitioning terminates once a best partitioning result is achieved. However, the GMP algorithm finds a partitioning result and improves the QoP of the result until the best result is found. During the partitioning process, it is guaranteed that the new partitioning result is always superior to the previous one. By controlling the desired degree of QoP, various partitioning results are obtained. For example, partitioning results of the GMP-baseline algorithm regarding the task tree shown in Figure 28.a, are presented in Figure 29. The partitioning result of the HIPART algorithm is equivalent to the partition result when the depth of tree is 2. When the depth is 3, the partition result is replaced by a fine-grain alternative. Figure 29 demonstrates that the partitioning result of the GMP-baseline algorithm improves as the depth of the tree increases. Thus, in this

example, various QoPs are attained by controlling or specifying the depth of tree generally. The QoP is not directly corresponded to the level of tree hierarchy.

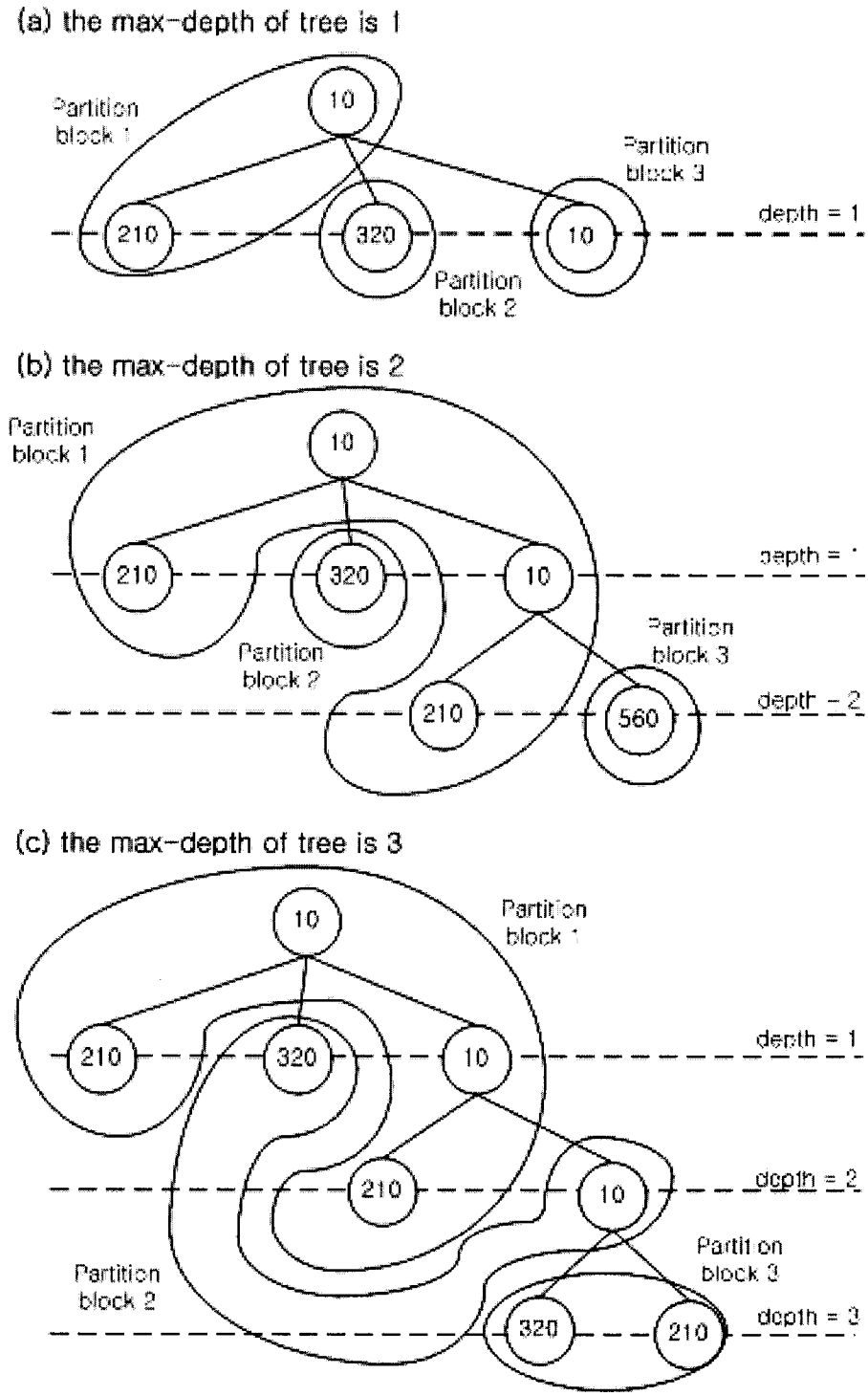


Figure 29. Various Partitioning Granularity of the GMP-Baseline Algorithm

CHAPTER 8. EXPERIMENTS

8.1 GMP Algorithm

The computational complexity analysis of the GMP-baseline algorithm is presented in chapter 4.3 and summarized in Table 6. In the general case, the complexity of the initial partitioning is described using the number of partition blocks, p, and the number of fan-out of a k-ary tree, k. The complexity of the E^2S partitioning algorithm is a function of the total number of node expansions, γ, the number of fan-out of a k-ary tree, k, and the average time required to identify *e-partitioning* that contains *e-node*, $\bar{r}$. In the worst case, the complexity of both algorithms is computed using the total number of atomic nodes in a cost tree, n, and the depth of the tree, d.

Table 6. The Computational Complexity of the GMP-baseline algorithm

$\delta(AMP_{base})$	Initial Partitioning	E^2S Partitioning	Complexity
General scenario	$k \cdot \left(\left\lceil \frac{p-k}{k-1} \right\rceil + 1 \right) + 1$	$\sum_{i=0}^{\gamma} \left(2 + \bar{r} + k\right)$	$O(n)$
Worst case	$1 + n^{1-\frac{1}{d}} + n$	$\frac{(n-1)\cdot(3+n^{1/d})}{n^{1/d}-1}$	$O(n)$

A set of initial partitioning results for various d, k, and n of a cost tree, $T(d,k,n)$, are presented in Figure 30. These results illustrate that the execution time of the GMP algorithm grows as fast as k^d for $T(d,k,k^d)$, and n for $T(d,*,n)$[1] with respect to d. As the depth of tree increases, the total number of atomic nodes grows proportionally. In the worst case analysis, k^d is equal to n.

[1] * means "not specified" or "don't care"

Figure 31 shows both the non-normalized and the normalized execution time of the initial partitioning algorithm for various d and k. The normalized execution time is obtained by dividing the non-normalized execution time by p. In both the non-normalized and the normalized version, the execution time of the initial partitioning algorithm rapidly decreases as p becomes larger except when p is 10. As shown in the normalized version, the execution time grows back after k is 6. It is because the optimal k for the given number of partition blocks, *10*, is 6 (i.e., $\lfloor \frac{10}{2} + 1 \rfloor$)

Figure 32 shows that the optimal value of k, opt_k, for the number of partition blocks, p, is equal to $\lfloor \frac{p}{2} + 1 \rfloor$ when k is greater than 1 and less than p. For example, opt_k is 7, 8,10, and 12 when p is 12, 15, 18, and 21, respectively. A set of E^2S partitioning results for various d, k, and n for a cost tree, $T(d,k,n)$ is presented in Figure 33. Similar to Figure 30, these results indicate that the execution time of the GMP algorithm increases as fast as k^d for $T(d,k,*)$, as n for $T(d,*,n)$, with respect to d. Since E^2S partitioning starts from the initial partitioning results, the number of partition blocks is not changed during the partitioning process.

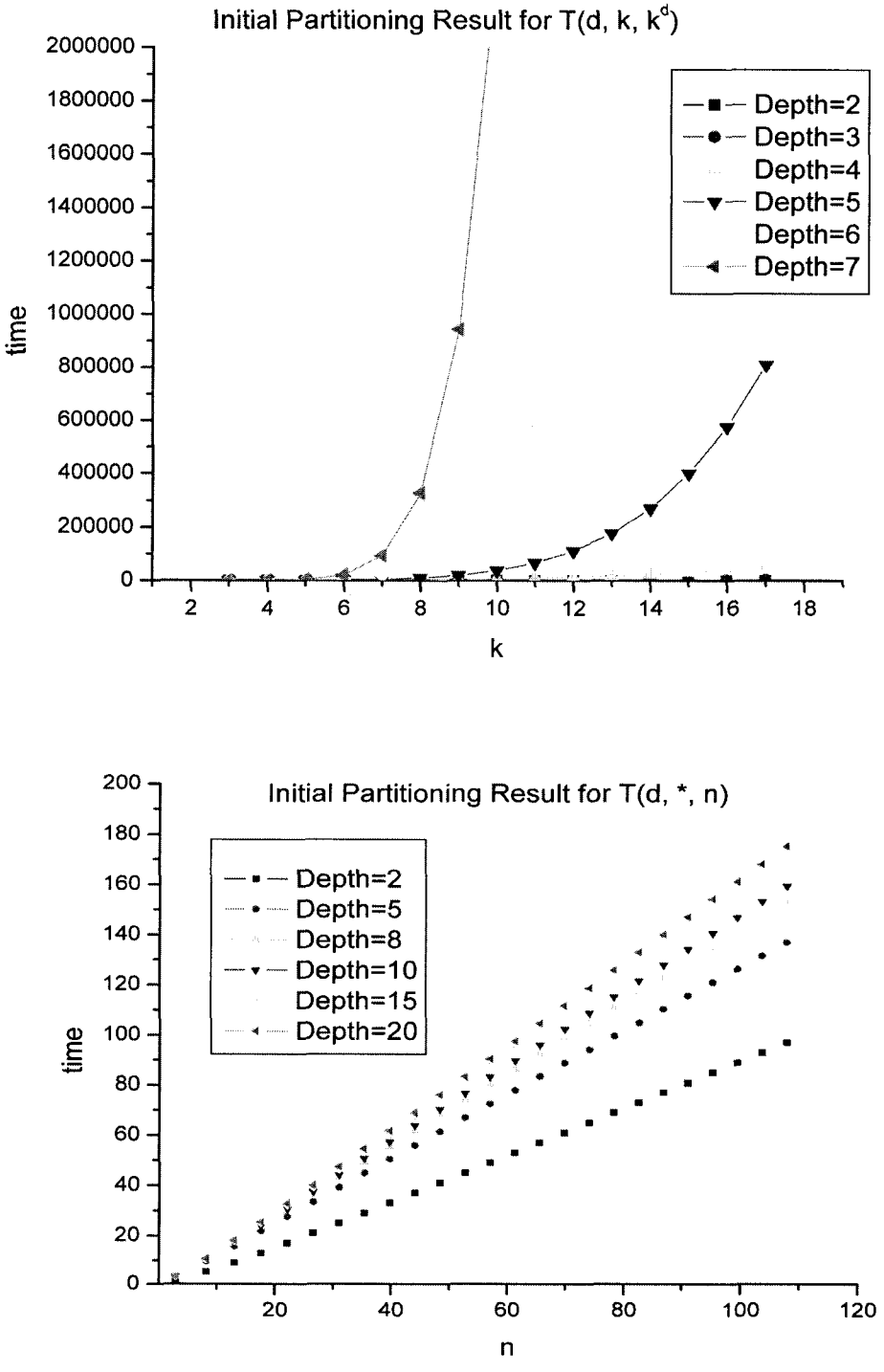


Figure 30. Initial Partitioning Results for T(d,k, d^k) and T(d,*,n)

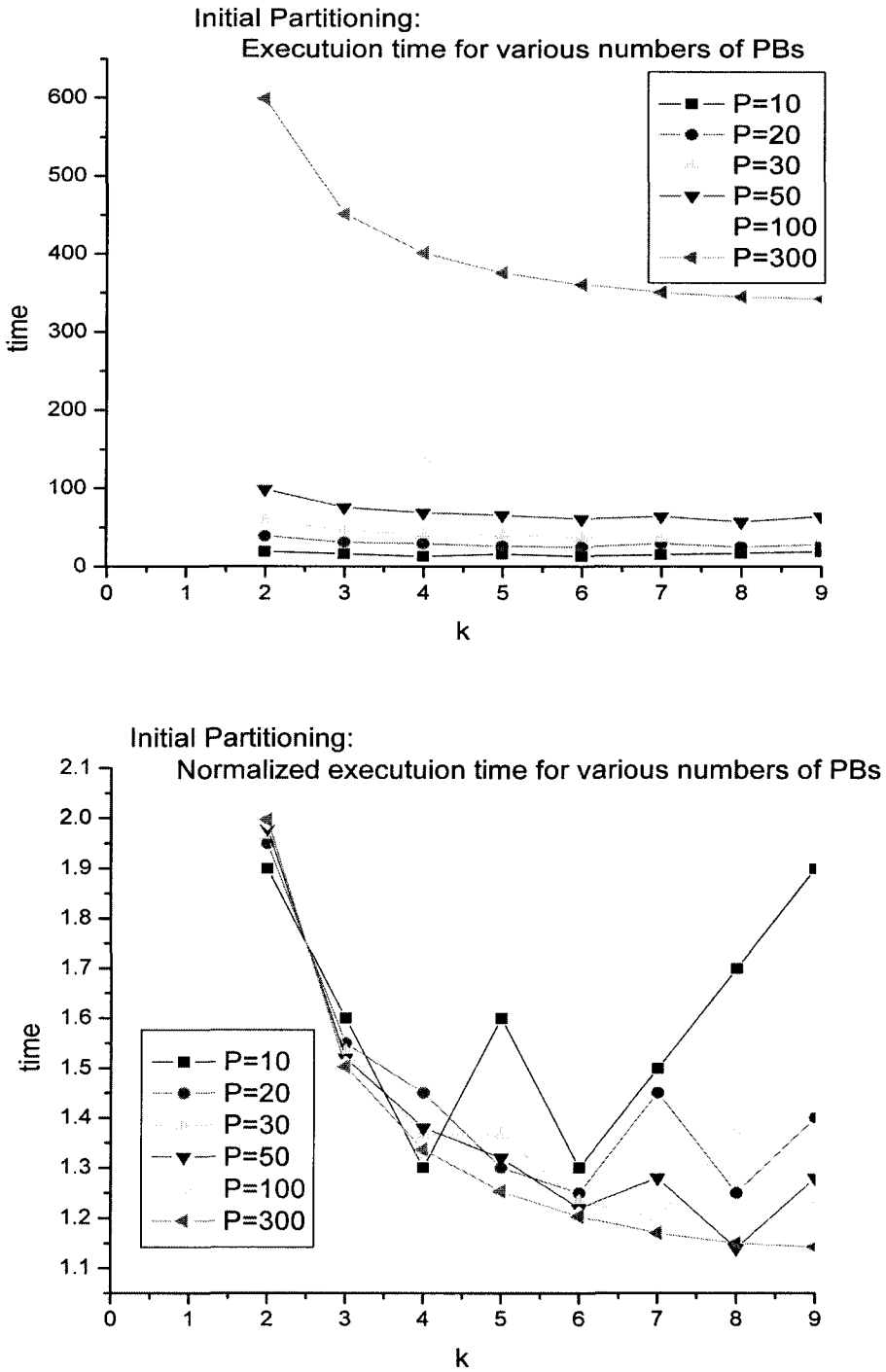


Figure 31. Initial Partitioning: Normalized versus Non-Normalized Execution time

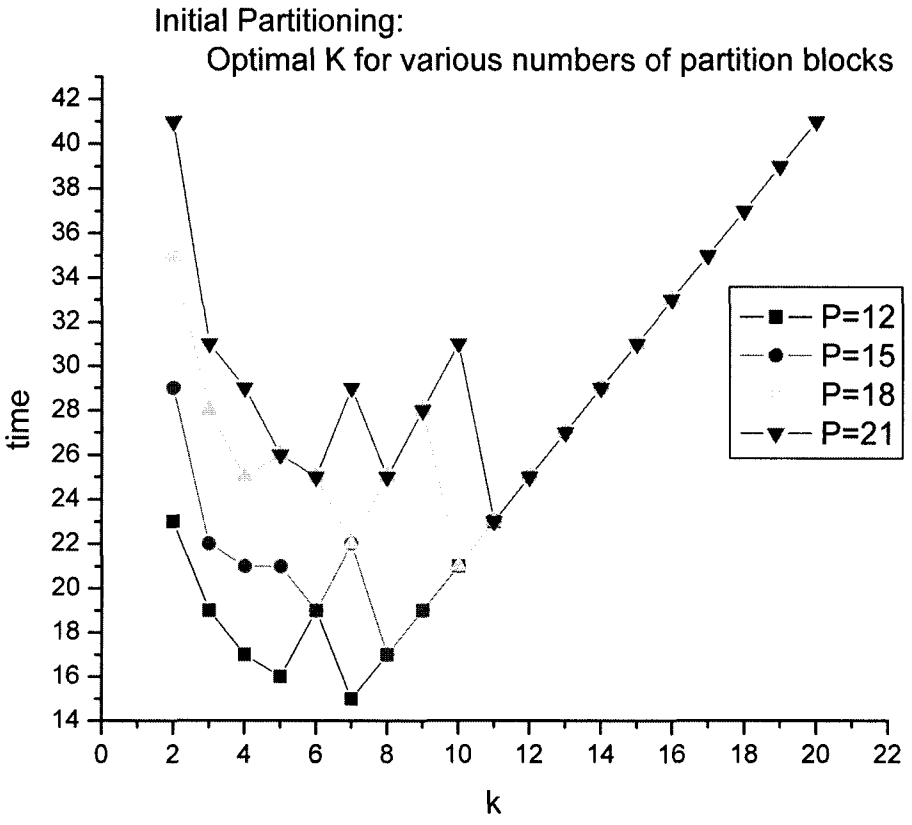
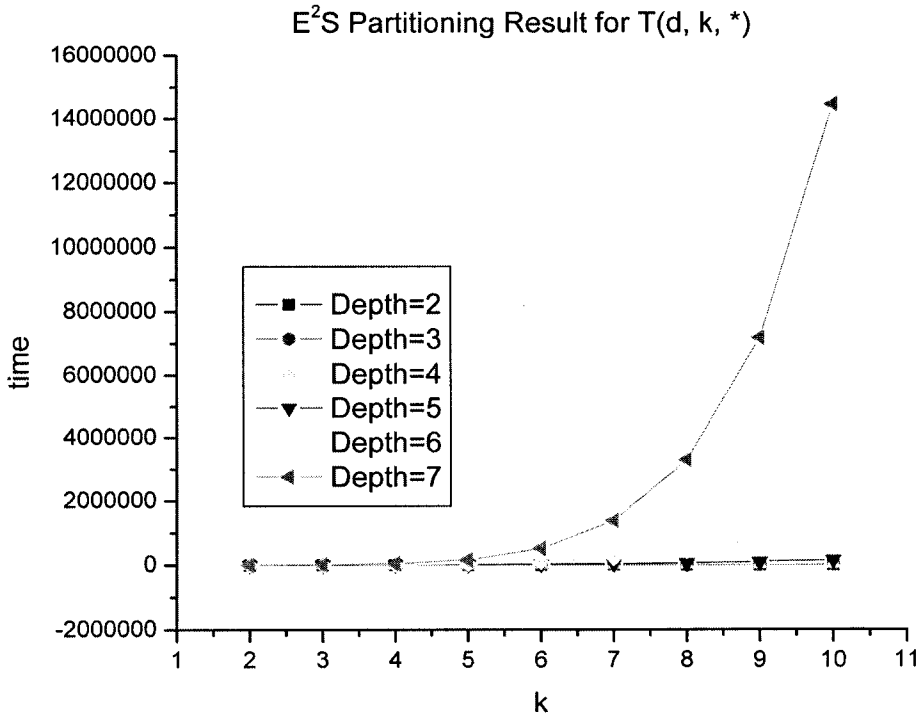


Figure 32. Initial Partitioning: Optimal k for Various Numbers of Partition Blocks



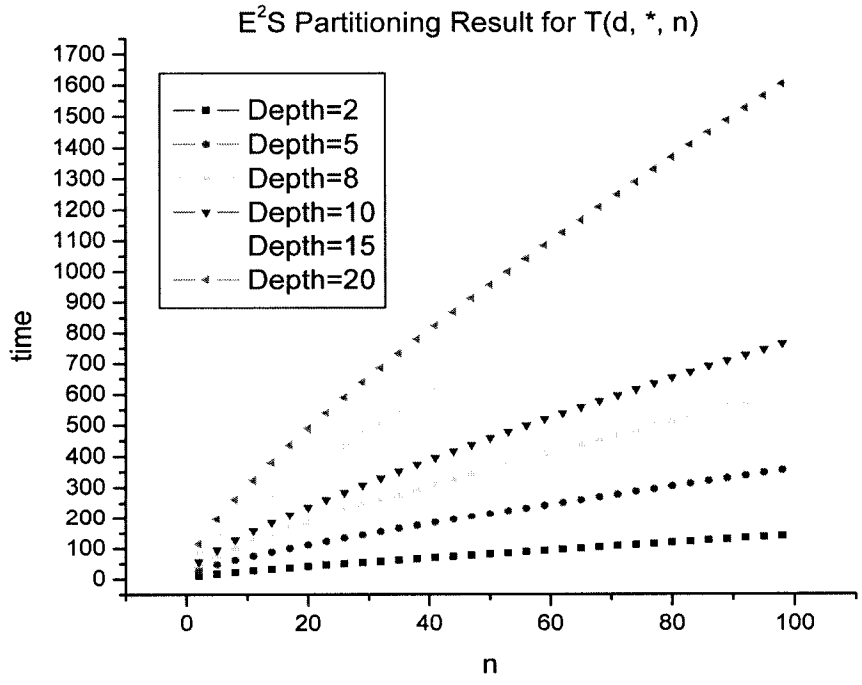


Figure 33. E²S Partitioning Results for T(d,k,*) and T(d,*,n)

8.2 Partitioning Result Evaluation

Figure 34 and Figure 35 show the cost evaluation results for a set of partition blocks using various cost evaluation measures. The effect of using two arithmetic cost evaluation measures, the *normalized cost* and the *cost difference,* and two statistical cost evaluation measures, the *cost distance from cost average* and the *cost variation,* are depicted in Figure 34 and in Figure 35, respectively. The normalized cost is computed by dividing the cost of a partition block by the cost of the partition block having the largest cost. The cost difference is calculated by adding all cost differences between a partition block and other partition blocks. Similarly, the cost distance from the cost average is obtained by measuring cost difference between the cost of a partition block and the cost average of all partition blocks. The cost variance is acquired by computing the square of difference between the cost of a partition block and the cost average.

These figures illustrate that numerous evaluation measures can be applied to evaluate partitioning results that are produced during the partitioning process. It is because the GMP algorithm does not depend on a specific technique to evaluate those results. This allows evaluation of the same result from different perspectives without the modification of the algorithm.

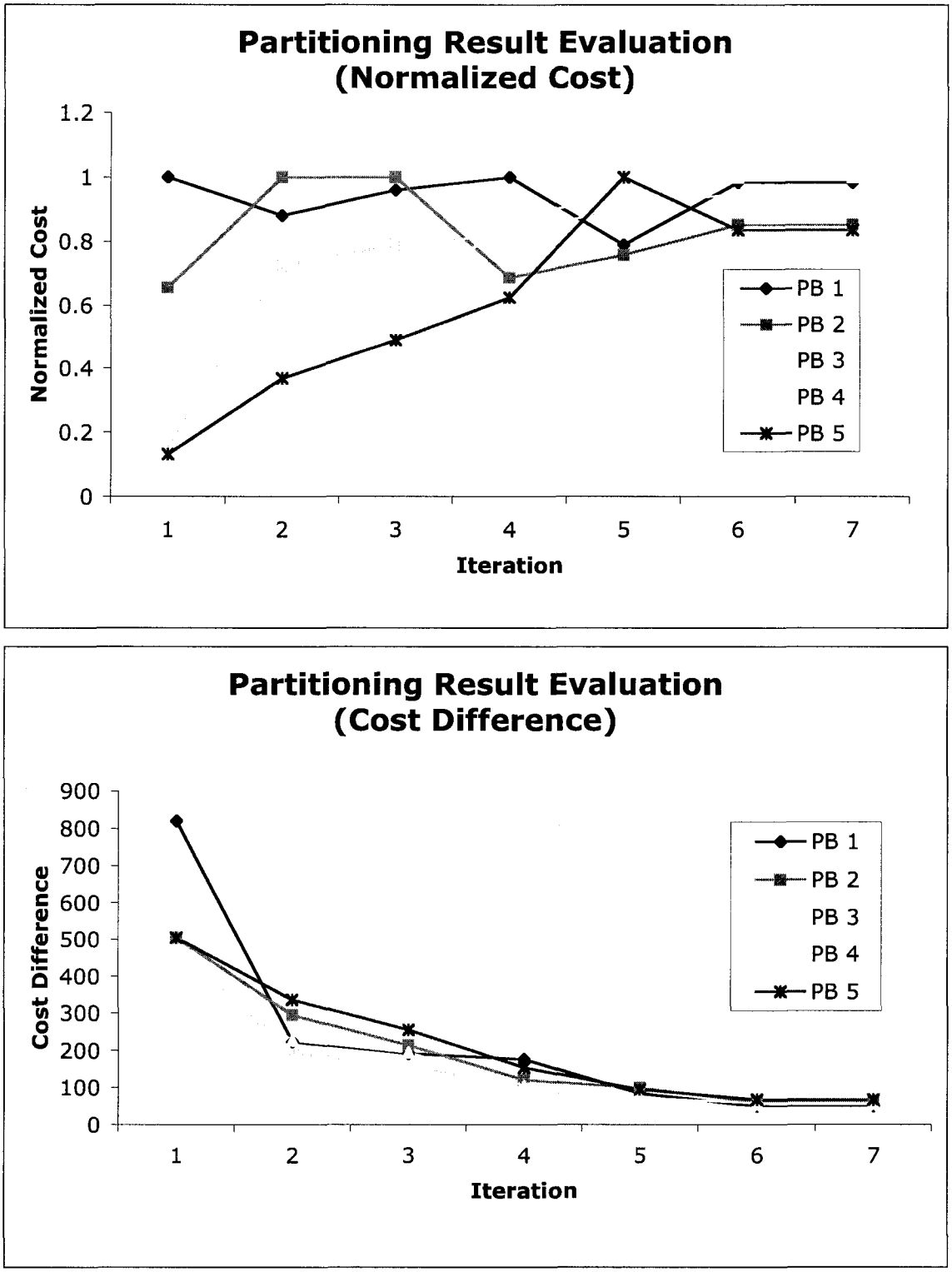


Figure 34. Partitioning Result Evaluation using Arithmetic Cost Evaluation Measures

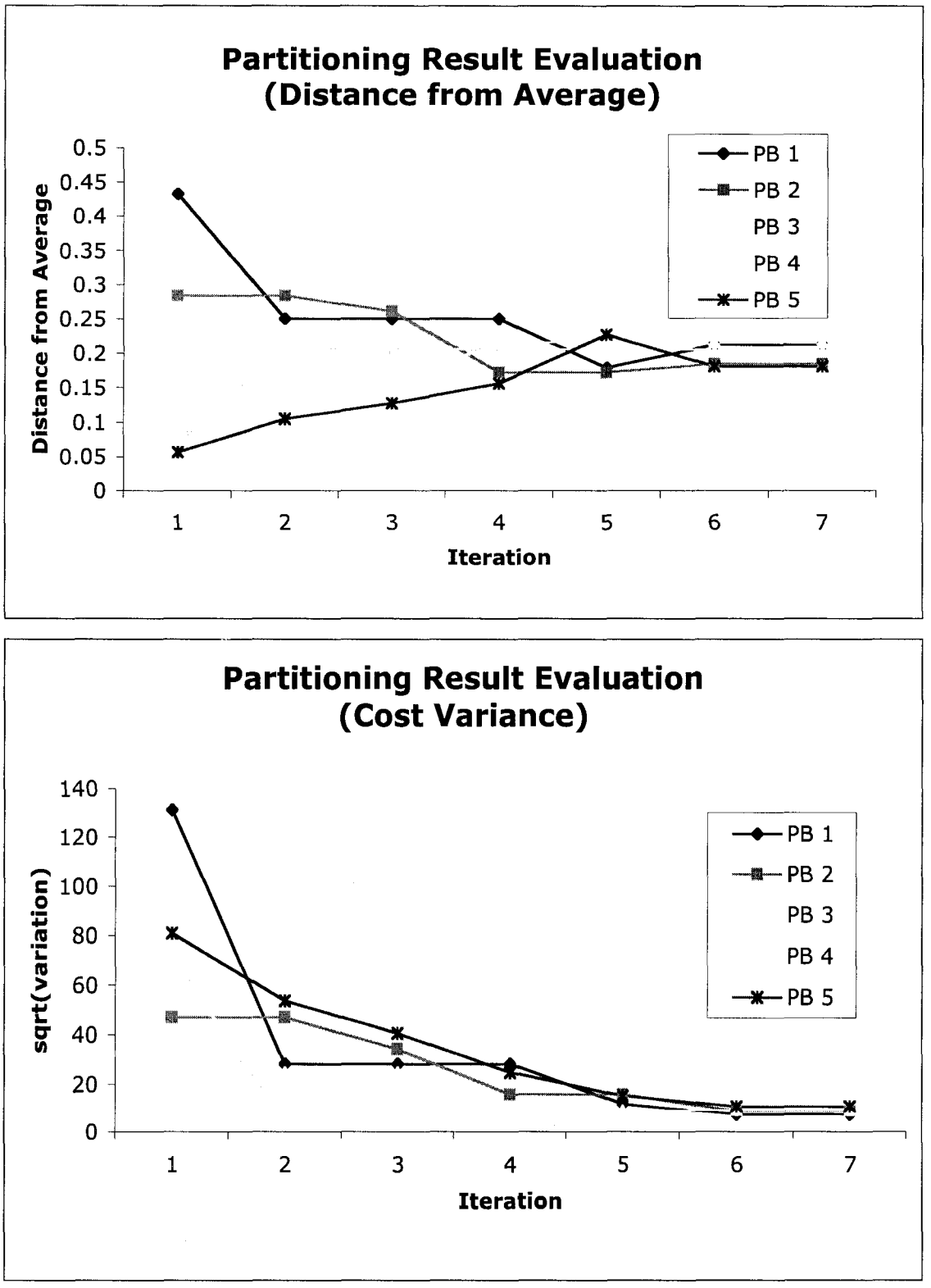


Figure 35. Partitioning Result Evaluation using Statistical Cost Evaluation Measures

8.3 Incremental QoP Improvement

Figure 36 shows incremental QoP improvements over various numbers of partition blocks, p. The cost disparity is used to evaluate the partitioning result. The cost disparity is reciprocal to the QoP. In the figure, the QoP is improved as the iteration count increases. Since the iteration count implies the node expansion in the cost tree, a higher iteration count means a longer execution time. The QoP is the best when p is 4 and is the worst when p is 13. From the perspective of the execution time, it takes the longest time to complete the partitioning task when p is 9 and the shortest time when p is 2. As p grows, it seems the total number of node expansion also increases. This is primarily because the cost tree, with respect to the given partitioning problem, expands to fill empty partition blocks during initial partitioning. If the root node of the cost tree contains as many as or more nodes than p, such expansions are eliminated. The interesting fact is that the QoP when p is 2 is superior to the QoP when p is 9. This demonstrates that a larger number of partitioning blocks does not always produce better partitioning results as compared to the results generated from a smaller number of partition blocks. One of main reasons for this is that the cost disparity between partition blocks generally grows as p increases. Another reason is that the GMP algorithm produces partition results having a high degree of the QoP when p is small. This is because a node of the cost tree represents all costs including the cost of the node and its all descendents if the node is a coupled node.

The figure illustrates that the QoP of partitioning results are improved without any degradation for any number of partition blocks. It also depicts that the QoP is directly

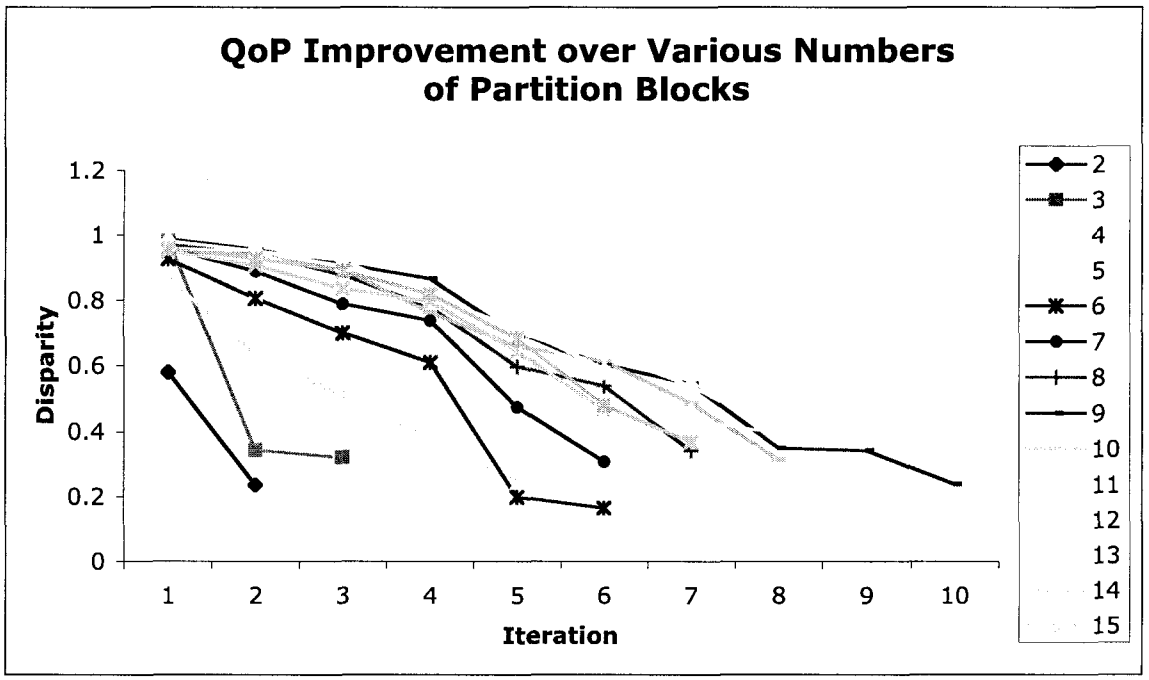


Figure 36. Incremental QoP Improvements over Various Number of Partition Blocks

related to neither the number of partition blocks nor the complexity of a given partitioning problem. Rather, it is associated with the cost information of the problem. Based on experiments regarding incremental QoP improvement, we can conclude that the proposed GMP algorithm is scalable and adaptable.

8.4 Experiments Using Various Cost Patterns

A series of experiments are conducted to evaluate the GMP algorithm as compared to other hierarchical partitioning algorithms from the perspectives of QoP and model execution time. A set of cost tree is randomly generated based on various cost patterns. Generated cost trees are partitioned into partition blocks by hierarchical partitioning algorithms. The quality of partitioning results is measured using quality measures. Also, the partitioned models are dispatched to a set of processors and their execution time is measured.

A set of cost trees is generated by a cost tree generator using various cost patterns for a given cost tree model. A cost tree model is specified by three parameters; the depth of the tree, d, the number of fan-outs of a coupled node, k, and the total number of atomic nodes of the tree, n. To preserve d depth with k fan-out, the n should be a number between $\sum_{i=1}^{d-1}(k-1)+k$ and k^d. Various cost patterns are generated based on Probability Mass Functions (PMF) shown in Table 7.

Table 7. Cost Patterns used in Cost Tree Generation [51-54]

Cost Pattern	PMF	Parameters	Distribution
$P_{unitstep}(x)$	$\delta(x)$[2]	None	Unit Step
$P_{uniform}(x)$	1	None	Uniform
$P_{exponential}(x)$	$\lambda e^{-\lambda x}$	$\lambda = 0.05$	Exponential
$P_{invgaussian}(x)$	$\sqrt{\frac{\lambda}{2\pi x^3}} e^{\frac{-\lambda(x-\mu)^2}{2\mu^2 x}}$	$\mu = 3.86, \lambda = 9.46$	Inverse Gaussian
$P_{parcto}(x)$	$\alpha k^{\alpha} x^{-\alpha-1}$	$\alpha = 1.245, k = 3$	Pareto
$P_{lognormal}(x)$	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$	$\mu = 5.929, \sigma = 0.321$	Lognormal

The GMP algorithm is compared to two other partitioning algorithms, random partitioning algorithm and ratio-cut partitioning algorithm, over various numbers of partition blocks and cost patterns. The random partitioning algorithm arbitrarily selects cost nodes and assigns them to a set of partition blocks. The ratio-cut partitioning algorithm cuts a sub tree that has the minimum cost disparity as compared to the average cost of a given cost tree. The average cost of the given cost tree is computed by dividing the cost of the root node of the tree by the requested number of partition blocks. Once a sub tree is assigned to a partition block, the average cost is recomputed with excluding the sub tree. This is repeated until only one partition block is left. The last partition block is populated with cost nodes that are not assigned to other partition blocks.

The experimental results over various numbers of partition blocks and cost patterns are presented in Figure 37 and Figure 38. The cost disparity and the average of cost differences between partition blocks are used as partitioning evaluation measures. In both

[2] $\delta(x)$ is a function that returns 1 when $x = a$. otherwise, returns 0.

figures, for a particular number of partition blocks and a cost pattern, each partitioning algorithm is applied to 20 different cost trees. The average of the partitioning results of the algorithm is computed and illustrated as a single mark in those figures.

Figure 37 shows the partitioning results of the GMP algorithm maintain the lowest cost disparity between partition blocks as compared to the results produced by other algorithms. In the GMP algorithm, the cost disparity between partition blocks is consistent regardless of cost patterns and the number of partition blocks. It is because the GMP algorithm minimizes the cost disparity by expanding cost nodes until the minimum cost disparity is obtained. In the random partitioning algorithm, the cost disparity between partition blocks decreases as the number of partition blocks increases. It is because the total number of node expansions increases when the number of partition blocks becomes large. The cost of a node is always larger than the costs of its descendents. Thus, as the total number of node expansions becomes larger, the cost disparity between cost nodes becomes smaller. This leads the cost disparity between partition blocks in the final partitioning result to be reduced. In the ratio-cut algorithm, the cost disparity between partition blocks arises until a certain number of partition blocks and declines afterward. In the algorithm, the cost disparity is highly associated with the cost difference between sub trees. As the number of partition blocks increases, the average cost difference between sub trees also increases. This makes cost disparity between partition blocks arises. However, after a certain number of partition blocks, the average cost decreases as the number of partition blocks increases. This reduces the cost disparity between partition blocks.

Figure 38 illustrates the GMP algorithm is also superior to other algorithms from the perspective of the average cost difference between partition blocks. In the random and the ratio-cut algorithm, the average cost difference arises until a certain number of partition blocks and declines afterward. However, the average cost difference in the GMP algorithm increases as the number of partition blocks becomes larger. The GMP algorithm maintains the cost disparity between partition blocks minimum based on available cost nodes when partitioning occurs. As the number of partition blocks increases, the average number of cost nodes per partition block decreases. This makes the cost disparity between partition blocks arises. However, the GMP algorithm still provides superior partitioning results as compared to other algorithms.

The partitioning results shown in both figures are summarized in **Table 8**. Each value in the table represents the average of all partitioning results associated with a particular cost pattern and a partitioning algorithm when the number of partition blocks varies from 2 to 50.

Table 8. The Summary of Partitioning T(5, 4, 50) using various Cost Patterns

	Cost Disparity			Average Cost Difference		
Cost Pattern	RANDOM	RATIO-CUT	GMP	RANDOM	RATIO-CUT	GMP
$P_{unitstep}$	14.0632653	16.8163265	2.71428571	38.4009438	35.7643095	19.6650933
$P_{uniform}$	759.4	596.70102	67.0510204	2543.64	1699.58667	582.160016
$P_{exponential}$	764.523469	590.404082	67.2040816	2538.18943	1691.82313	583.6735
$P_{invgaussian}$	758.591837	587.433673	67.0397959	2525.80815	1681.96659	578.869121
P_{pareto}	763.421429	592.528571	67.394898	2539.75575	1695.76506	582.621707
$P_{lognormal}$	755.004082	588.485714	66.622449	2527.40202	1685.60103	578.898615

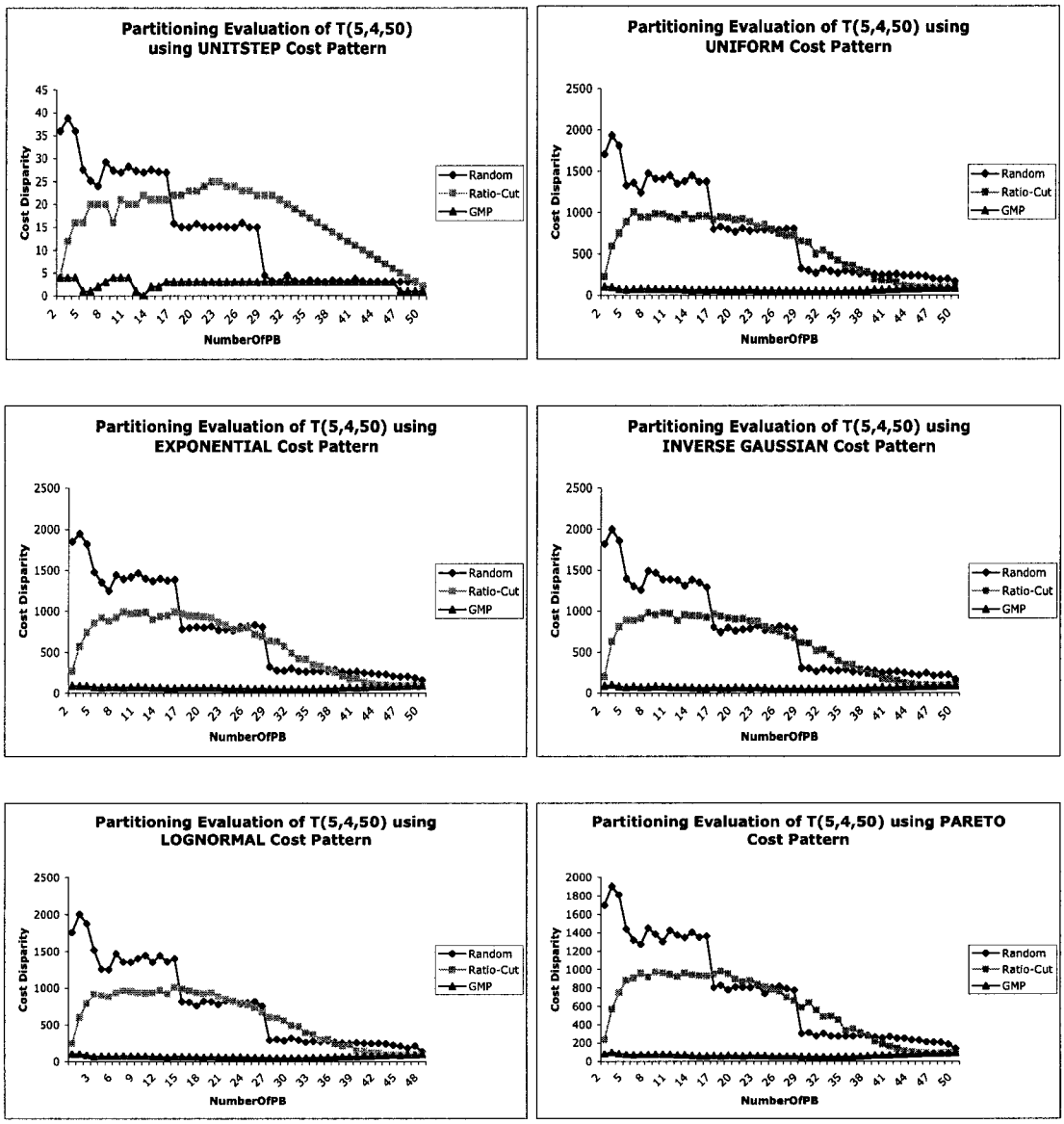


Figure 37. Partitioning Evaluation of T(5,4,50) using Cost Disparity

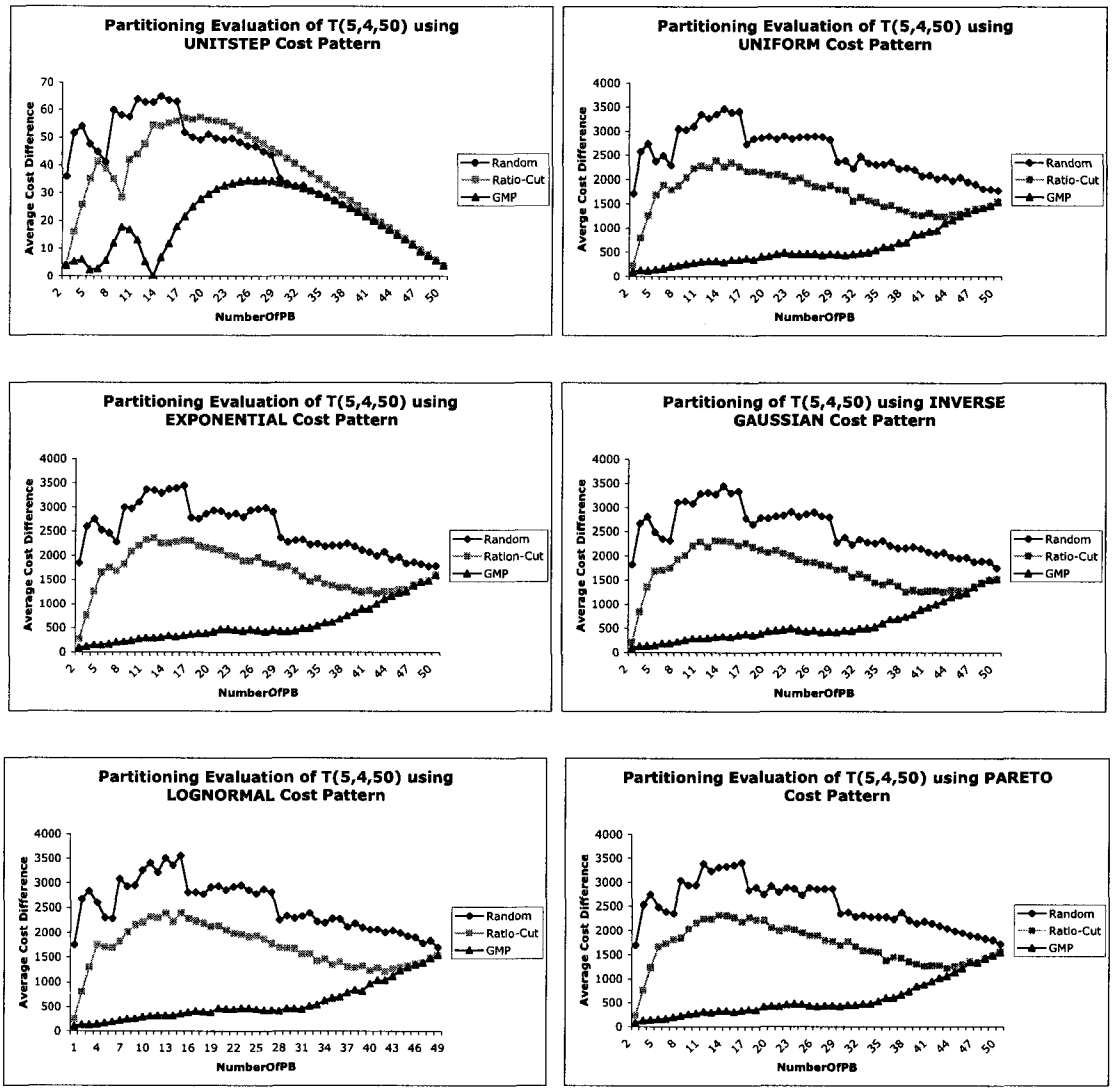


Figure 38. Partitioning Evaluation of T(5,4,50) using Average Cost Distance

The experimental results on the execution time of the partitioned models are presented from Figure 40 to Figure 44. The experiments are started by generating a set of cost trees regarding various numbers of partition blocks, of processors, and cost patterns. The generated cost trees have same structural organization, T(4,3,52), but, contains different cost information. The cost information is determined by cost patterns that are applied when a series of cost values are produced by the cost tree generator. The cost patterns used in the experiments are UNITSTEP, UNIFORM, EXPONENTIAL, INVERSE GAUSSIAN, and PARETO. Once a cost tree is built, the tree is partitioned by the three partitioning algorithms based on the given number of partition blocks, of processors, and a cost pattern. The partitioning blocks produced by each partitioning algorithm are dispatched to a set of processors. Upon the partition blocks are received by processors, models in the partition blocks are executed concurrently and their execution time is collected. The total execution time of the partitioned models is determined by identifying the model that takes the longest execution time. The GMP algorithm provides the shortest execution time in most experiments as compared to other algorithms. And, the GMP algorithm also shows consistent execution time over various numbers of partition blocks and cost patterns. Figure 39 and Table 9 show the experimental results when the number of partition blocks is equal to the number of available processors. Figures from Figure 40 to Figure 45 and Table 10 show the experimental results when the number of partition blocks is equal to or larger than the number of processors.

The random partitioning algorithm, in general, produces the shortest execution time when the number of partition blocks is considerably large. However, the algorithm

generates longer execution time when the number of partition blocks is either small or medium as compared to the GMP algorithm. The execution time of the algorithm is highly sensitive to the number of partition blocks. As the number of partition blocks becomes larger, the total number of node expansions increases. The growth of node expansions reduces cost difference between cost nodes. This allows producing the partition result having smaller cost disparity. The execution time of the algorithm fluctuates regardless of the number of partition blocks and cost patterns because cost nodes are randomly selected and assigned to partition blocks. Thus, it is difficult to determine generic execution time characteristic of the algorithm.

The ratio-cut partitioning algorithm produces consistent execution time over various numbers of partition blocks. Whenever a node is assigned to a partition block, the average cost of the given cost tree is recomputed. This allows the cost disparity between partition blocks to be minimized. However, the algorithm produces the longest execution time in most cases as compared to other algorithms. It is because the execution time is related to the cost disparity between cost nodes rather than between partition blocks. The first partition block is filled with a cost node having the smallest cost disparity with respect to the average cost of the original cost tree and the last partition block is populated with remaining cost nodes that are not included in other partition blocks. Thus, the cost disparity between the node in the first partition block and the node having the smallest cost in other partition blocks is considerably large. Since the cost in this experiment represents activity of a model, higher activity means longer execution time.

Thus, the ratio-cut algorithm takes longer execution time as compared to other algorithms.

The GMP algorithm produces consistent execution time over various numbers of partition blocks and runs faster than other algorithms when the number of partition blocks is either small or medium and as fast as the random partitioning algorithm when the number of partition blocks is large. This is achieved by permitting the child nodes of a cost node to be assigned to separate partition blocks, if necessary, to minimize the cost disparity between partition blocks during partitioning process. This leads the GMP algorithm to produce partition blocks that contain the smaller cost disparity and run faster than other algorithms.

In a single processor, the partitioned models that are generated by GMP algorithm and the ratio-cut algorithm run in the shortest time and in the longest time, respectively. In the random partitioning, the execution time radically decreases as the number of partition blocks increases. As the number of partition blocks becomes larger, the total number of node expansions occurs more frequently and the cost difference between cost nodes becomes smaller. This reduces the cost disparity between partition blocks of the final partitioning result. In the ratio-cut partitioning, the execution time remains consistent over various number of partition blocks because the cost disparity between models having the largest cost and the smallest cost is less sensitive to the number of partition blocks. Thus, the partitioned models generated by the ratio-cut algorithm take longer execution time as compared to the models that are produced by other algorithms. Especially, when the number of partition block is large.

In multiple processors, the partitioned models that are produced by the GMP algorithm run faster than the models produced by other algorithms in most cases. As the multiple numbers of processors are involved in model execution, in general, the model execution time is reduced in every partitioning algorithm. However, the partitioned models that are produced by random partitioning algorithm occasionally run slower on multiple processors. It is owing to the nondeterministic nature of the random partitioning algorithm. Regardless of the number of processors, the algorithm produces randomly created partitioning results. The execution time of the partitioned models that are generated by the ratio-cut algorithm is significantly reduced as multiple processors are involved in model execution. The models produced by the ratio-cut algorithm runs faster than the models produced by the random algorithm when the number of partition blocks is small. However, the models produced by the ratio-cut algorithm runs still slower than the models created by the random algorithm when the number of partition blocks is large.

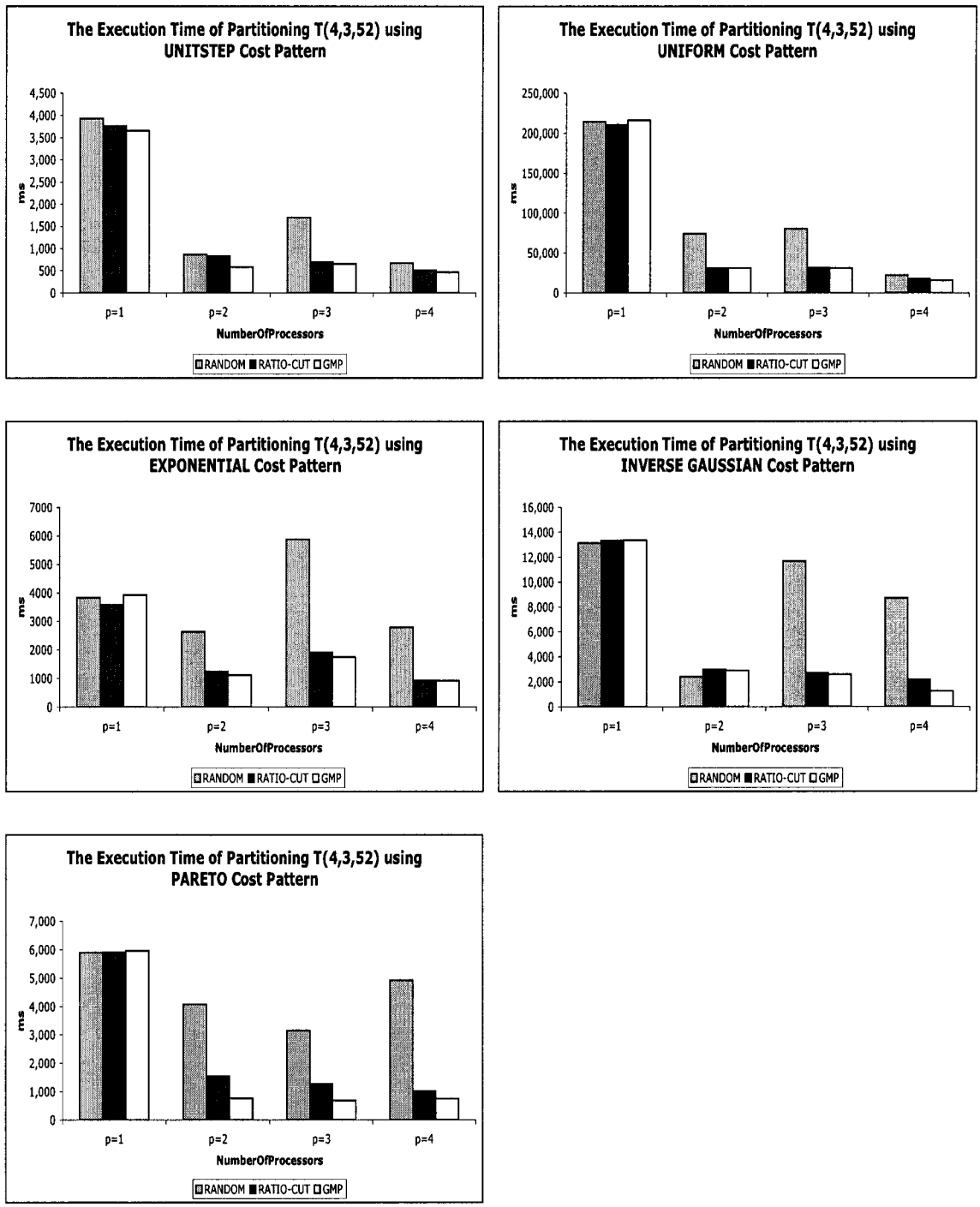


Figure 39. The Execution Time of Partitioning Tree T(4,3,52) When the Number of Partition Blocks is Equal to the Number of Available Processors

Table 9. The Average Execution Time When the Number of Partition Blocks is Equal to the Number of Available Processors

Number of Processor	Cost Pattern	Partitioning Algorithm		
		RANDOM	RATIO-CUT	GMP
1	UNITSTEP	3,929	3,766	3,656
	UNIFORM	214,746	210,592	216,029
	EXPONENTIAL	3833	3591	3923
	INV_GAUSSIAN	13,130	13,333	13,363
	PARETO	5,898	5,911	5,969
2	UNITSTEP	862	831	581
	UNIFORM	74668	31225	31125
	EXPONENTIAL	2634	1252	1121
	INV_GAUSSIAN	2414	2967	2915
	PARETO	4086	1543	761
3	UNITSTEP	1702	694	652
	UNIFORM	80726	31537	31089
	EXPONENTIAL	5883	1903	1741
	INV_GAUSSIAN	11689	2703	2576
	PARETO	3155	1272	687
4	UNITSTEP	671	505	461
	UNIFORM	22132	17545	15811
	EXPONENTIAL	2794	931	911
	INV_GAUSSIAN	8745	2202	1244
	PARETO	4926	1021	756

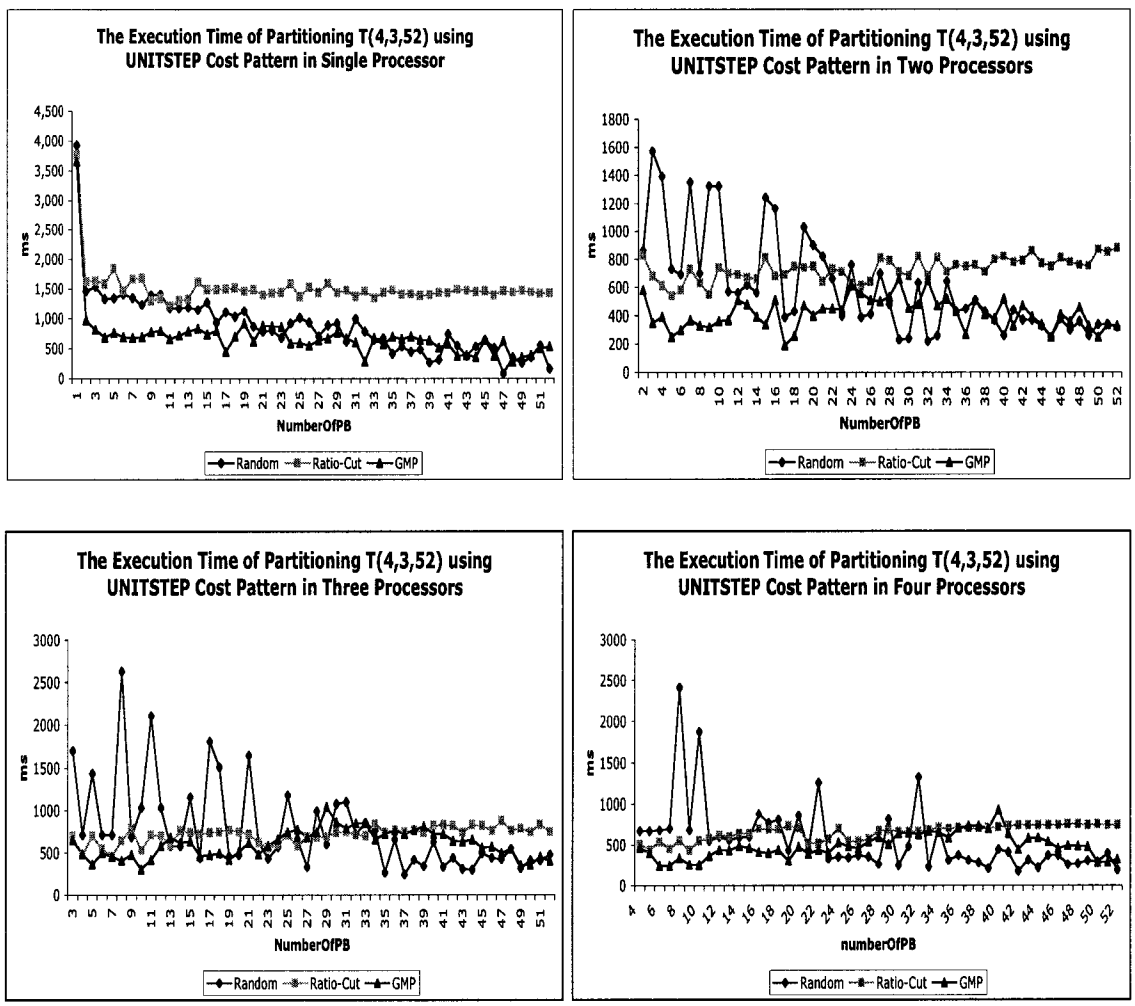


Figure 40. The Execution Time of Partitioning Tree T(4,3,52) using UNITSTEP Cost Patterns on Various Numbers of Processors

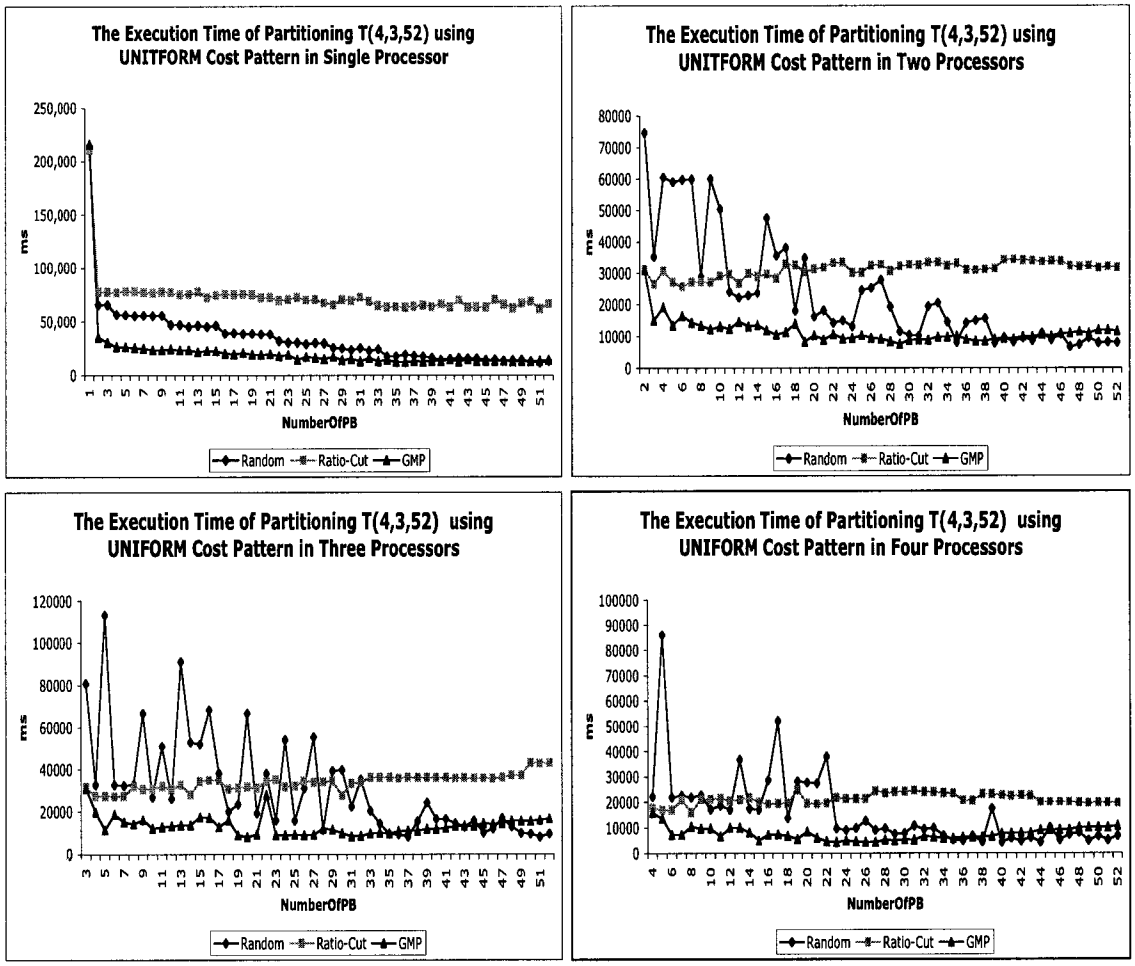


Figure 41. The Execution Time of Partitioning Tree T(4,3,52) using UNIFORM Cost Patterns on Various Numbers of Processors

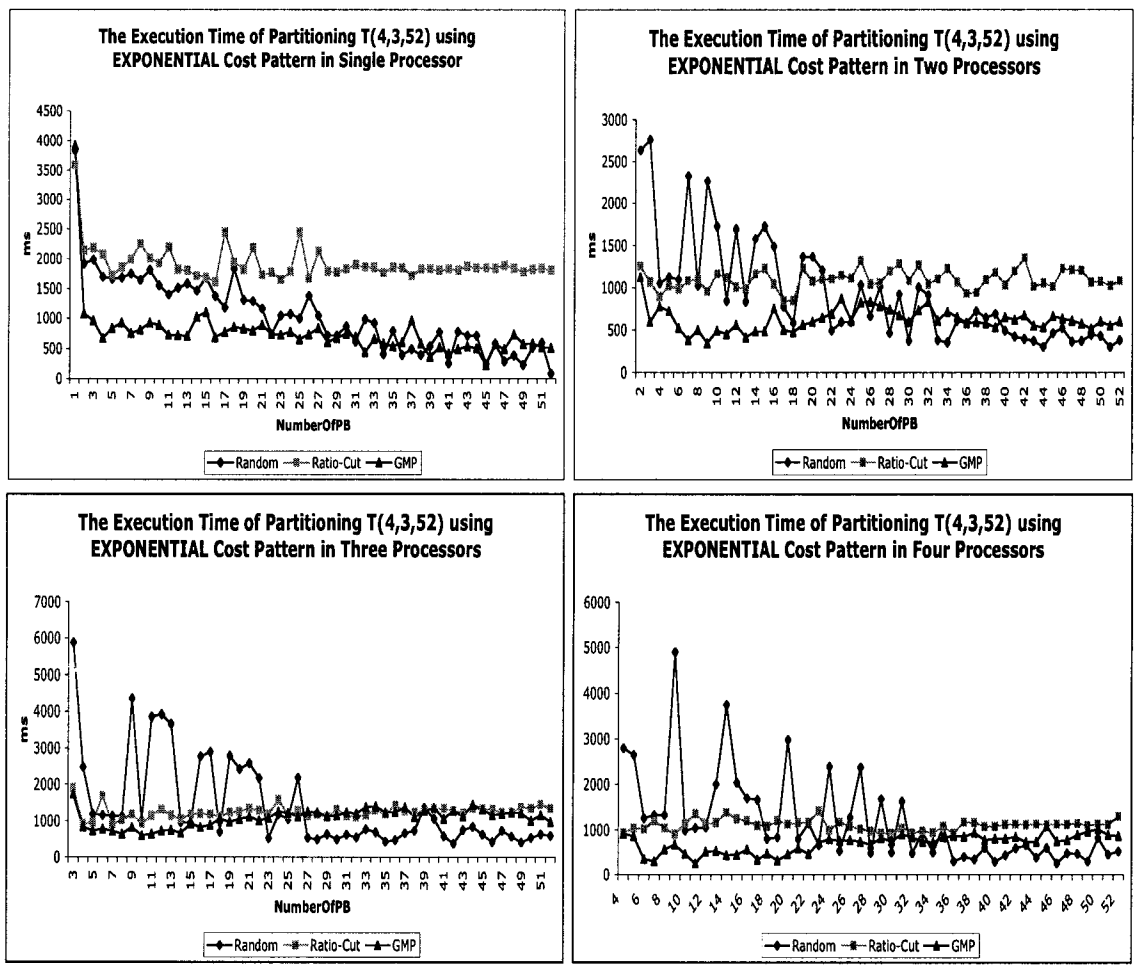


Figure 42. The Execution Time of Partitioning Tree T(4,3,52) using EXPONENTIAL Cost Patterns on Various Numbers of Processors

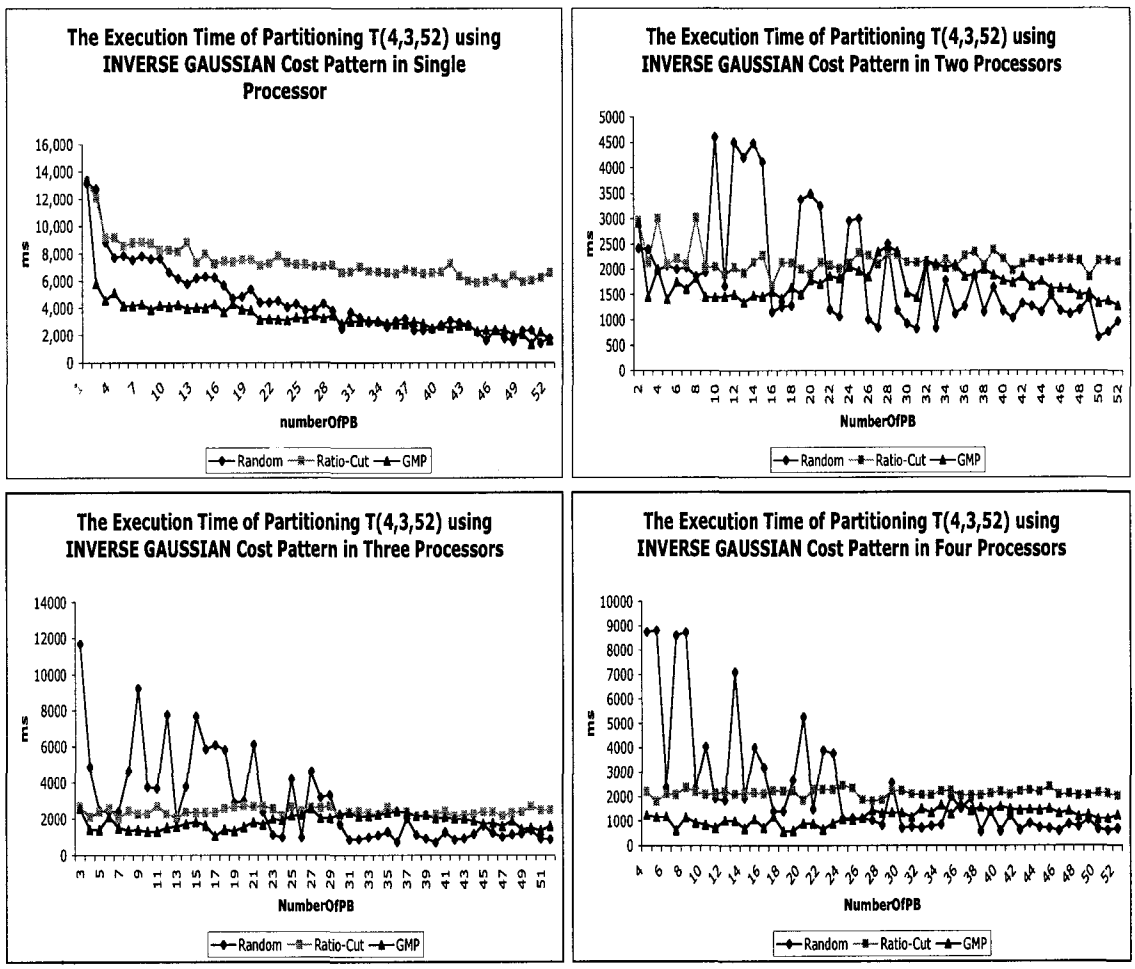


Figure 43. The Execution Time of Partitioning Tree T(4,3,52) using INVERSE GAUSSIAN Cost Patterns on Various Numbers of Processors

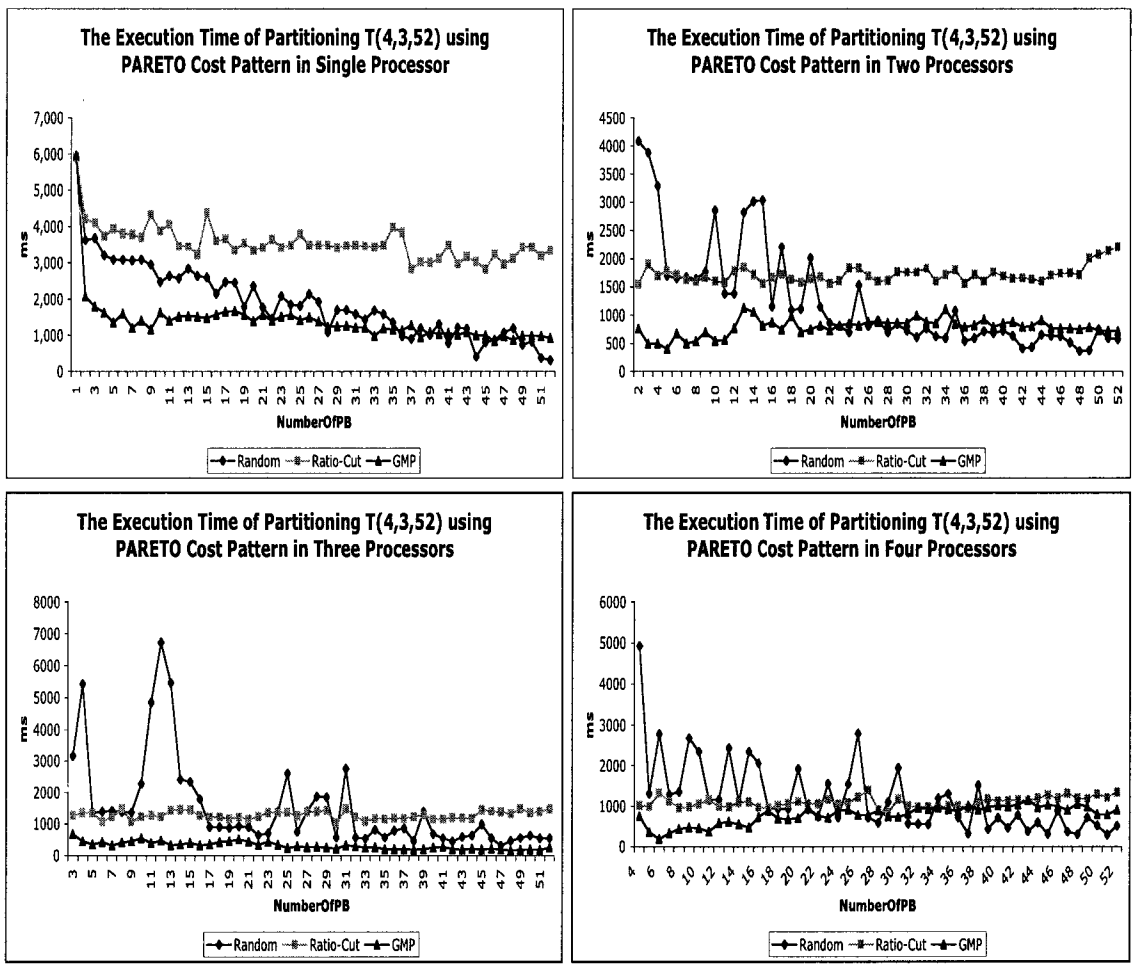


Figure 44. The Execution Time of Partitioning Tree T(4,3,52) using PARETO Cost Patterns on Various Numbers of Processors

Table 10. The Average Execution Time over Various Numbers of Processors

Number of Processor	Cost Pattern	Partitioning Algorithm		
		RANDOM	RATIO-CUT	GMP
1	UNITSTEP	892	1,510	699
	UNIFORM	34,788	72,930	21,785
	EXPONENTIAL	1,070	1,919	761
	INV_GAUSSIAN	4,555	7,396	3,467
	PARETO	1,903	3,546	1,390
2	UNITSTEP	604	731	417
	UNIFORM	23,347	31,248	11,401
	EXPONENTIAL	915	1,093	618
	INV_GAUSSIAN	1,885	2,174	1,746
	PARETO	1,257	1,715	786
3	UNITSTEP	775	717	604
	UNIFORM	31,050	33,915	13,363
	EXPONENTIAL	1,390	1,244	1,079
	INV_GAUSSIAN	2,854	2,399	1,833
	PARETO	1,481	1,285	316
4	UNITSTEP	558	652	489
	UNIFORM	15,653	21,105	7,566
	EXPONENTIAL	1,173	1,101	697
	INV_GAUSSIAN	2,293	2,136	1,166
	PARETO	1,190	1,096	785

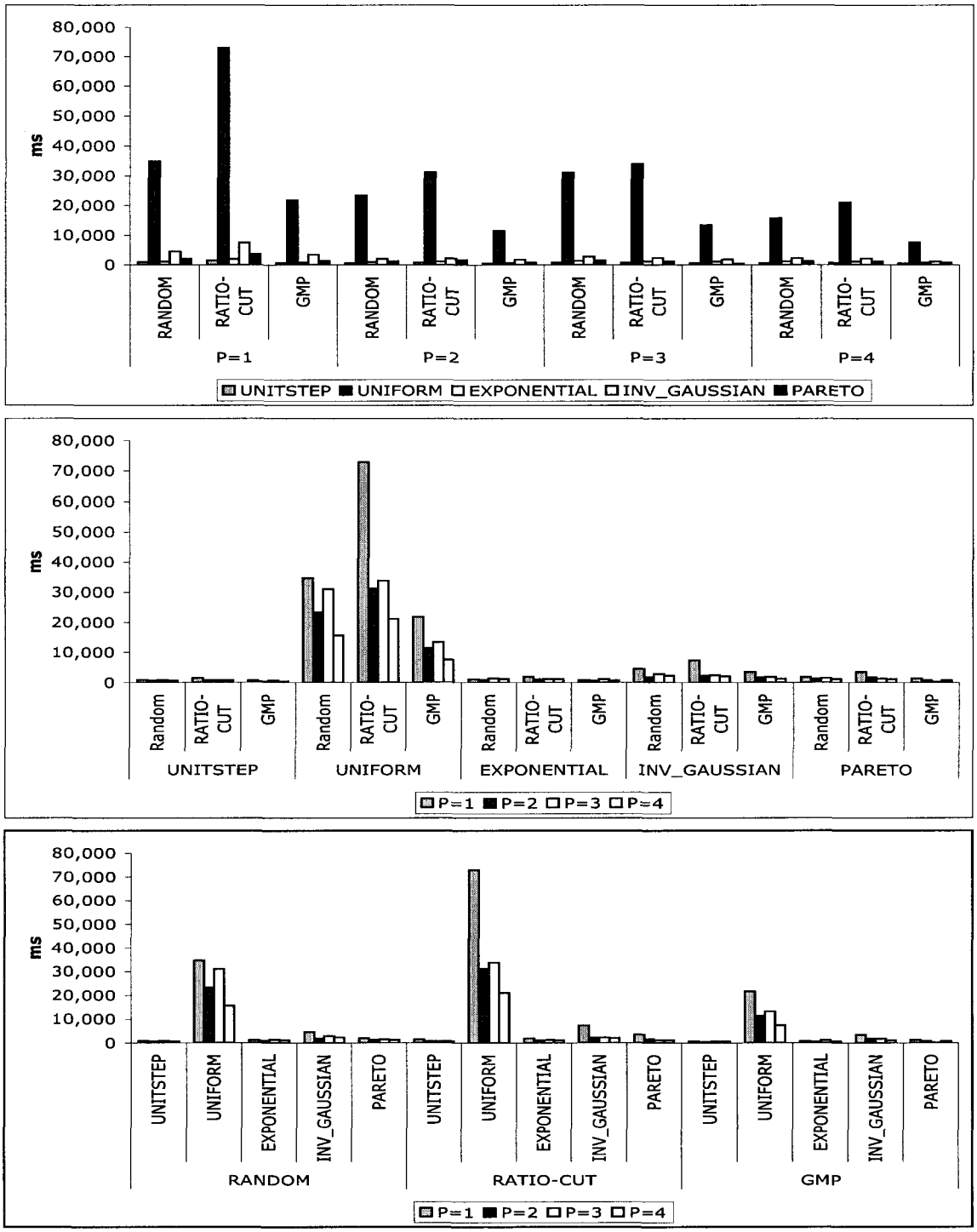


Figure 45. The Summary of Experimental Results

CHAPTER 9. CONCLUSION

The main objective of this research is to design and implement a class of generic partitioning algorithms for hierarchical, modular DEVS models for distributed simulation. To attain this goal, a set of partitioning algorithms is designed based on the cost analysis methodology. The methodology leads to GMP algorithms that are concise, generic, and adaptable.

The cost analysis methodology is briefly discussed in CHAPTER 3. Major issues regarding the proposed partitioning algorithms are discussed from the perspective of cost analysis. A series of cost measures for cost generation, cost evaluation, and cost aggregation are presented. By applying cost measures, the proposed algorithms enable capturing and manipulating heterogeneous properties of given models with the homogenous measure, *cost*. Since a cost measure is a parametric method, subject to certain axioms, the proposed algorithms are generic and applicable to any family of models, provided there is a way to manipulate the appropriate cost information.

A new Generic Model Partitioning (GMP) algorithm is proposed in CHAPTER 4. The GMP algorithm decomposes a hierarchical model into a set of partition blocks and provides solutions for distinct partitioning problems based on the cost analysis methodology. It also minimizes model decomposition during the partitioning process and guarantees an incremental Quality of Partitioning (QoP) property that always produces increasingly improved partitioning results until a desired partitioning solution is attained. Partitioning improvement that occurred during the partitioning process is captured in the

partitioning tree and is easily tractable through the tree hierarchy. Theoretical analysis of the algorithm is followed by a discussion of optimality issues in various aspects.

In CHAPTER 5, a class of advanced algorithms derived from the GMP algorithm is presented to show its extendibility and adaptability for various distributed system configurations. The GMP algorithms for homogeneous and heterogeneous systems based on static and dynamic information are discussed (i.e., GMP-SHM, GMP-SHT, GMP-DHM, and GMP-DHT). The GMP-lookahead(k) algorithm is derived from the GMP-baseline algorithm, introduced in CHAPTER 4, to show its versatility as a GMP-SHM algorithm. Furthermore, an adaptive model partitioning algorithm (AMP) derived from the GMP-baseline algorithm is presented to show how it could be applied to implement GMP-SHT, GMP-DHM, and GMP-DHT algorithms.

Existing and possible new applications of the GMP algorithms are discussed in CHAPTER 6. As an existing example, DEVS/ADNS is introduced. DEVS/ADNS is a DEVS M&S framework targeting advanced network infrastructures such as P2P and GRID. The GMP algorithms are used to implement major components of DEVS/ADNS (i.e., model partitioner and model deployer). As a possible example, static/adaptive resource allocator based on the GMP algorithms is presented. Unlike conventional resource allocators, the allocator could handle both hierarchical and non-hierarchical resource allocation requests. Also, the N-body mapping problem is introduced to show the usability of the GMP algorithms in the field of scientific computing.

The proposed GMP algorithms are compared to existing DEVS model partitioning algorithms in CHAPTER 7. The comparison shows GMP algorithms could emulate their

results, and provide better flexibility and adaptability for many partitioning problems. Various experimental results on GMP algorithms are presented in CHAPTER 8.

9.1 Future Research

There are a few issues that are not discussed in previous chapters. Those issues include *hierarchical model partitioning based on a deepening approach*, *hierarchical model partitioning using a dynamic cost tree*, and *full-scale survey of partitioning optimality associated with various cost tree topologies*. It is desirable to perform further research on those issues as the part of the study on cost-based partitioning for distributed simulation of hierarchical, modular DEVS models.

Through further research on those issues, it is expected that the GMP algorithm deals with a spectrum of hierarchical model partitioning problems based on both decomposition and deepening approaches. Also, by supporting partitioning for a given dynamic cost tree, it is anticipated that the proposed algorithm solves various sophisticated partitioning problems that exhibit dynamic structural permutations over time and space (e.g., the N-body problem). Furthermore, it is predicted that the optimal number of partition blocks for a given partitioning problem is identified based on survey results obtained from research on correlation between cost tree topologies and partitioning optimality.

REFERENCES

[1] A. Pothen, "Graph Partitioning Algorithms with Applications to Scientific Computing," *Parallel Numerical Algorithms*, pp. 323-368, 1997.

[2] P. Fjallstrom, "Algorithms for Graph Partitioning: A Survey," *Computer and Information Science*, vol. 3, 1998.

[3] A. Frieze and M. Jerrum, "Improved approximation algorithms for MAX k-CUT and MAX BISECTION," *Alogorithmica*, vol. 18, pp. 61-77, 1994.

[4] M. R. Banan and K. D. Hjelmstad, "Self-organization of architecture by simulated hierarchical adaptive random partitioning," presented at International Joint Conference of Neural Networks (IJCNN), 1992.

[5] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graph," *The Bell System Technical Journal*, vol. 49, pp. 291-307, 1970.

[6] V. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671-680, 1983.

[7] M. J. Berger and S. H. Bokhari, "A Partitioning Strategy for Non-Uniform Problems across Multiprocessors," *IEEE Transactions on Computers*, vol. 36, pp. 570-580, 1987.

[8] H. D. Simon, "Partitioning of Unstructured Problems for Parallel Processing," *Computing Systems in Engineering*, vol. 2, pp. 135-148, 1991.

[9] Y. Z. a. G. Karypis, "Evaluation of Hierarchical Clustering Algorithms for Document Datasets," *CIKM 2002*, 2002.

[10] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. 5, pp. 440-452, 1979.

[11] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digitial Systems," *IEEE Transactions on Computers*, vol. 33, pp. 160-177, 1984.

[12] D. Jefferson and H. Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism," presented at The Society for Computer Simulation(SCS), La Jolla, California, 1985.

[13] D. A. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404-425, 1985.

[14] B. P. Zeigler, H. Praehofer, and T. Kim, *Theory of Modeling and Simulation*, 2 ed. San Diego: Academic Press, 2000.

[15] ACIMS, "DEVSJAVA Modeling and Simulation Package," ACIMS, University of Arizona, http://www.acims.arizona.edu/SOFTWARE/software.shtml, 2003.

[16] B. P. Zeigler and H. Sarjoughian, "Introduction to DEVS Modeling and Simulation with JAVA: A Simplified Approach to HLA-Compliant Distributed Simulations," http://www.aicms.arizona.edu, 2001.

[17] J. Ameghino, A. Tróccoli, and G. Wainer, "Models of Complex Physical Systems using Cell-DEVS," presented at Proceedings of the Annual Simulation Symposium, Seattle, Washington, 2001.

[18] A. Muzy, E. Innocenti, A. Aiello, J.-F. Santucci, and G. Wainer, "Cell-DEVS Quantization Techniques in a Fire Spreading Application," presented at Winter Simulation Conference, San Diego, California, 2002.

[19] J. Nutaro, B. P. Zeigler, R. Jammalamadaka, and S. Akerar, "Discrete Event Solution of Gas Dynamics within the DEVS Framework:Exploiting Spatiotemporal Heterogeneity," presented at Internation Conference on Computational Science, Melbourne, Australia, 2003.

[20] B. P. Zeigler and G. Ball, "Bandwidth Utilization/Fidelity Tradeoffs in Predictive Filtering," presented at Simulation Interoperability Workshop, Orlando, Florida, 1999.

[21] B. P. Zeigler and H. Cho, "Quantization Based Filtering in Distributed Discrete Event Simulation," vol. 62(11), pp. 1629-1647, 2002.

[22] ACIMS, "DEVS M&S Software Packages," *ACIMS*, http://www.acims.arizona.edu/SOFTWARE/software.shtml , 2003.

[23] B. P. Zeigler and G. Ball, "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions," presented at Simulation Interoperability Workshop, Orlando, Florida, 1999.

[24] Y. Cho, B. P. Zeigler, and H. Sarjoughian, "Design and Implementation of Distributed Real-Time DEVS/CORBA," *IEEE System, Man, Cybernatics Conference*, 2001.

[25] O. M. Group(OMG), *CORBA/IIOP Specification*: OMG, http://www.omg.org/technology/documents/formal/corba_iiop.htm, 2002.

[26] O. M. Group(OMG), *OMG IDL / Language Mappings Specifications*: OMG, http://www.omg.org/technology/documents/idl2x_spec_catalog.htm, 2003.

[27] DMSO, "High Level Architecuture," https://www.dmso.mil/public/transition/hla, 2003.

[28] DMSO, "Runtime Infrastructure (RTI)," https://www.dmso.mil/public/transition/hla/rti, 2003.

[29] S. Cheon, "Design and Implementation of Distributed Scalable DEVS M&S Framework over Peer-to-Peer Network System," in *The Department of Electrical and Computer Engineering*: M.S. Thesis, University of Arizona, 2002.

[30] D. A. Menasce and H. Gomma, "A method for design and performance modeling of client/server systems," *IEEE Transactions on Software Engineering*, vol. 26, pp. 1066-1085, 2000.

[31] D. Clark, "Face-to-Face with Peer-to-Peer Networking," in *Computer*, vol. 1, 2001, pp. 18-21.

[32] M. H. Alsuwaiyel, *Algorithms Design Techniques and Analysis*, vol. 7. Danvers: World Scientific Publishing, 1999.

[33] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Algorithms*. Reading, MA: Addison-Wesley, 1975.

[34] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. NewYork: McGraw-Hill, 1992.

[35] R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*: Addison-Wesley, 1996.

[36] M. Sipser, *Introduction to the Theory of Computation*. Boston: PWS, 1997.

[37] D. E. Knuth, "Big Omicron and big Omega and big Theta," *SIGACT News*, pp. 18-24, 1976.

[38] D. E. Knuth, *The Art of Computer Programming*, vol. 1, 2 ed: Addison-Wesley, 1998.

[39] D. E. Knuth, *The Art of Computer Programming*, vol. 2, 2 ed: Addison-Wesley, 1998.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*: Addison Wesley, 1994.

[41] D. G. Childers, *Probability and Random Processes*: McGraw-Hill, 1997.

[42] S. Ross, *Stochastic Processes*, 2 ed. NewYork: John Wiley & Sons, 1996.

[43] A. O. Allen, *Probability, Statistics, and Queing Theory with Computer Science Applications*, 2 ed. San Diego: Academic Press, 1990.

[44] R. L. Graham, D. E. Knuth, and O. Pthashnik, *Concrete MatheMatics: A Foundation for Computer Science*: Addison-Wesley, 1994.

[45] I. Foster, C. Kesselman, and S. TUecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *The International Journal of Supercomputer Applications*, vol. 15, 2001.

[46] J. Project, "JXTA Project," http://www.jxta.org.

[47] I. Foster and C. Kesselman, "Glolbus: A metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, pp. 115-128, 1997.

[48] J. E. Barnes and P. Hut, "A Hierarchical *O(NlogN)* force calculation algorithm," *Nature*, vol. 324, pp. 446-449, 1986.

[49] K. Kim, T. Kim, and K. Park, "Hierarchical partitioning algorithm for optimistic distributed simulation of DEVS models," *Journal of Systems Architecture*, vol. 44, pp. 433-455, 1998.

[50] G. Zhang and B. P. Zeigler, "Mapping Hierarchical Discrete Models to Multiprocessor System: Concepts, Algorithm, and Simulation," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 271-281, 1990.

[51] V. Cardellini, M. Colajanni, and P. S. Yu, "Request Redirection Algorithms for Distributed Web Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 355-368, 2003.

[52] S. Floyd and V. Paxson, "Difficulties in Simulating the Internet," *IEEE/ACM Transactions on Networking*, vol. 9, pp. 392-403, 2001.

[53] M. F. Arlitt and T. Jin, "A Workload Characterization Study of the 1998 World Cup Web Site," *IEEE Network*, vol. 14, pp. 30-37, 2000.

[54] P. Barford and M. E. Crovella, "A Performance Evaluation of Hyper Text Transfer Protocols," *Proceeding of ACM Sigmetrics*, pp. 188-197, 1999.