COST-BASED PARTITIONING FOR DISTRIBUTED AND PARALLEL SIMULATION OF DECOMPOSABLE MULTI-SCALE CONSTRUCTIVE MODELS*

SUNWOO PARK†, C. ANTHONY HUNT†‡, AND BERNARD P. ZEIGLER§

Abstract. We present a concise, generic, and configurable new partitioning approach for decomposable, modular, and multi-scale constructive models. A Generic Model Partitioning (GMP) algorithm decomposes a given multi-scale model to a set of partition blocks based on a cost modeling and analysis method in polynomial time. It minimizes model decompositions and constructs monotonically improved partitioning outcomes during the partitioning process. The cost modeling and analysis method enables translating subjective, domain-specific, and heterogeneous resource information to objective, domain-independent, and homogeneous cost information. By translating models to a homogeneous cost space and describing partitioning logics over the space, the proposed algorithm utilizes domain-specific knowledge to produce the best partitioning results without any modification of its programming logics. As a consequence of its clean separation between domain specific partitioning requirements and goals, and generic partitioning logic, the proposed algorithm can be applied to a variety of partitioning problems in large-scale systems biology research utilizing distributed and parallel simulation. It is expected that the algorithm improves overall performance and efficiency of in silico experimentation of complex multi-scale biological system models.

Key words. Multi-scale partitioning, Systems biology, Model decomposition, DEVS, Multi-scale modeling and simulation

1. Introduction. Given the challenges faced by the emerging new field of systems biology [1–3], multi-scale constructive simulation modeling is an attractive approach for describing large, complex, multi-scale biological systems. It is expected to enable representing aspects of structural and behavioral characteristics of a multi-scale system hierarchies of components interacting with each other and their environment. Heterogeneous and multifaceted system features can also be represented within such models. Such Aggregations are often infeasible or difficult for the more traditional equation-based inductive models.

However, efficient execution of complex multi-scale simulation models is challenging. The models are easily exposed to low degrees of parallelism and are also prone to non-superlative resource distribution to a set of computational entities (e.g., processors) in distributed and parallel computing environments. In order to increase the degree of parallelism while optimizing resource allocation and managing core M&S issues, we need to consider computational and resource management issues. Prominent among these issues are model partitioning, model deployment, remote activation, parameter sweeping and optimization, and experimentation automation [4, 5]. Model partitioning constructs a set of fine-grain component models from a coarse-grain multi-scale model. Model deployment dispatches the decomposed models to the

*This research has been supported in part by CDH Research Foundation R21-CDH-00101, NSF DMI-0122227, and DOE SciDAC DE-FC02-01ER41184. We are grateful for the Computational and Systems Biology Postdoctoral Fellowship funding provided to Sunwoo Park by the CDH Foundation. The preliminary version of this paper was presented at the Challenges of Large Applications in Distributed Environments (CLADE) 2003, International Workshop on Heterogeneous and Adaptive Computation, June 2003.

†BioSystems Group, Dept. of Biopharmaceutical Sciences, University of California, San Francisco, 513 Parnassus Ave., San Francisco, CA 94143-0446 (spark4@itsa.ucsf.edu)

‡Joint Graduate Group in Bioengineering, University of California, Berkeley and San Francisco, 513 Parnassus Ave., San Francisco, CA 94143-0446 (hunt@itsa.ucsf.edu)

§Department of Electrical and Computer Engineering, University of Arizona, 1230 E. Speedway Blvd., Tucson, AZ 85721 (zeigler@ece.arizona.edu)

set of computational entities based on a certain heuristic. Remote activation reactively launches a simulator with a model and builds communication channels with other simulators when the model is available within the simulator's computational boundary. Parameter sweeping and optimization minimizes exploration of uninterested parameter spaces. Experimentation automation pipelines a series of distinctive experimental phases to an automated workflow for the efficient execution of a large-scale in silico experiment or large numbers of distinctive but repetitive experiments. Among those issues, this paper focuses on the issue of partitioning.

Multi-scale model partitioning plays a key role in efficient execution of complex multi-scale simulation models. By decomposing a complex multi-scale model to a set of component models, it enables building and maintaining optimal model distribution over computational entities and enhances the degree of parallelism. Design and implementation of generic partitioning algorithms that can be applied to a variety of multi-scale models is challenging. We must simultaneously consider two design aspects, *specialization* and *generalization*. It is desirable to use domain-specific or domain-aware knowledge to produce optimal partitioning results. However, existing partitioning algorithms use domain-independent or domain-neutral low-level information such as execution time, communication time, delay, and memory requirements. In doing so, it is preferable to maintain generic partitioning logics that can be widely applied. However, these considerations can conflict. We adopt a cost modeling and analysis method for our partitioning algorithm in order to reduce conflicts. The method enables translating domain-specific and heterogeneous resource information into objective, domain-independent, and homogeneous cost information. The method's use leads to a class of algorithms that efficiently partitions a set of decomposable multi-scale models while preserving important aspects of both the specialization and generalization paradigms. The concept and related issues of this method are addressed in section 3.

We propose a Generic Model Partitioning (GMP) algorithm that uses the cost modeling and analysis method to decompose a modular, multi-scale constructive model into a set of partition blocks in polynomial time. The algorithm describes a partitioning programming logic over a domain-independent cost space that is constructed by applying selected cost modeling and analysis techniques. The process allows the GMP algorithm to be concise, generic, and configurable. The algorithm produces high-quality partitioning outcomes with the minimum model decomposition. The Quality of Partitioning (QoP) is progressively improved until the best partitioning result is attained. Furthermore, it enables implementing various partitioning strategies by using different cost measures and functions instead of modifications of the partitioning logics. The algorithm is described detail in section 4. Complexity and execution time analysis of the algorithm are presented in the following section.

To present the usability and power of the GMP algorithm, we apply it to multi-scale, decomposable, modular Discrete Event Systems Specification (DEVS) models. DEVS is a discrete-event oriented multi-scale, constructive M&S approach [6–8]. It provides a solid foundation for theoretical or practical M&S driven systems biology and has been applied to multi-scale biological problems [9–12]. A set of GMP DEVS partitioners has been successfully developed for large-scale distributed simulation systems [4, 13, 14]. A collection of qualitative and quantitative experimental results and their analysis are presented in section 6.

2. Background and Related Work. Model partitioning is the process of aggregating or dividing (decomposable) models into a set of partition blocks. In dis-

tributed and parallel simulation systems, it plays vital roles in three processes: resource allocation and management, load sharing and balancing, and optimization. Performance, efficiency, and utilization can be significantly improved by optimally distributing models into active or passive system entities (e.g., simulators and coordinators). Optimal distribution is closely related to how models are partitioned and deployed. Thus, it is important to develop algorithms that produce optimal or, at least, acceptable partitioning results with respect to the end points of interest. However, most model partitioning algorithms focus on non-decomposable models that are formulated as a graph or a hyper-graph structure [15–18]. Multi-level partitioning algorithms transform the structure into a hierarchical alternative [19–23]. Neither deal with decomposable models. We can produce better partitioning results [4, 24, 25]. As model complexity increases they have naturally evolved into hierarchical and modular structures. Such evolution escalates the demand for new classes of partitioning algorithms that efficiently handle those structures.

Partitioning algorithms are mainly divided into three main classes: *random partitioning, partitioning refinement,* and *heuristic*. The random partitioning algorithms randomly aggregate or segregate models to a set of partition blocks. The partitioning refinement algorithms improve partitioning results during the partitioning process. The heuristic partitioning algorithms utilize domain-specific knowledge or particular optimization techniques.

The Kernighan-Lin (KL) algorithm is an example of random partitioning combined with partitioning refinement. The KL algorithm initially builds a partitioning result by randomly assigning models to partition blocks; it then revises the quality of the results by swapping models between those blocks whenever swapping produces a better partitioning result [26]. The performance of the KL algorithm has been improved from $O(n^3)$ to $O(max\{E \cdot log\, n, E \cdot deg_{max}\})$ by Dutt, and to $O(E)$ by Fiduccia and Mattheyeses, and to $O(V + E)$ by Diekmann, Monien, and Preis [27–29]. V, E, and deg_{max} are the total number of vertices, the total number of edges, and the maximum node degree, respectively. The quality of partitioning is substantially bound to the initial partitioning result. Thus, it is desirable to incorporate domain specific heuristics to improve result quality [30].

Multifarious heuristics have been applied to model partitioning algorithms. Structural and spatial relationships between models are used in recursive bisection algorithms. The algorithms split a graph into two sub-graphs and recursively bisect each sub-graph based on particular geometric information. Recursive coordinate bisection (RCB), recursive inertial bisection (RIB), and orthogonal recursive bisection (ORB) algorithms use the property of spatial orthogonality - a coordinate axis, axis of angular momentum, and an orthogonal plane to the axis [31–33]. Recursive graph bisection (RGB) algorithms use the shortest path length between two graph nodes [34]. Recursive Spectral Bisection (RSB) and eigenvector recursive bisection (ERB) algorithms use an eigen vector representing connectivity and distance between nodes [35–39]. Various optimization techniques including simulated annealing (SA), mean field annealing (MFA), tabu Search (TS), and genetic algorithm (GA) have been also applied to model partitioning algorithms [40–44].

Hierarchical partitioning works by either decomposing or building hierarchical structures based on specified decision-making criteria. Hierarchical structure is commonly represented by a multi-level, acyclic, directed graph (ADG) or a tree structure. During the partitioning process, the hierarchical structure is dynamically created and updated over time and space. A partitioning policy specifies how and when the struc-

ture is updated. Three widely used policies are *flattening*, *deepening*, and *heuristic*. Flattening is a structural decomposition technique that transforms the hierarchical structure into a non-hierarchical structure. Deepening, also known as hierarchical clustering, is a structural aggregation technique that transforms a non-hierarchical structure into a hierarchical one. Heuristic is any technique other than flattening and deepening. In this paper, we refer to partitioning algorithms based on the flattening and deepening approaches as multi-scale and multi-level partitioning algorithms, respectively. Multi-level partitioning has been investigated extensively over the past few decades [19–23]. However, multi-scale partitioning has received less attention.

In this paper, we reduce the scope of multi-scale partitioning algorithms to *random*, *ratio-cut*, and *heuristic*. For a given hierarchical and decomposable tree representing the homomorphic structure of a DEVS coupled model, a random algorithm decomposes the tree and randomly assigns nodes or subtrees to a set of partition blocks. A ratio-cut algorithm cuts a sub tree that has the minimum cost disparity as compared to the average cost of the tree. The average cost of the 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 the remaining nodes that are not assigned to other partition blocks. The HIPART algorithm is an example of the ratio-cut algorithm [24]. A heuristic algorithm is one that uses any technique other than random and ratio-cut approaches. The ENCLOSURE algorithm is an example of the heuristic algorithm [25].

3. Cost Modeling and Analysis Method. The cost modeling and analysis method provides a means of transforming heterogeneous *resource* information into homogeneous *cost* information whilie conducting analyses over a cost space. A *cost* is a homogeneous object representing heterogeneous resource information (e.g., single value, a set of discrete objects, and a continuous range). A cost *measure* is a conceptual metric that captures heterogeneous resource information in terms of cost (e.g., complexity, I/O connectivity, dynamic activity, and latency). Because a cost measure is a parametric method subject to certain axioms, algorithms based on the method are generic and applicable to any family of computational tasks (e.g., constructive simulation models) provided there is a way to manipulate the appropriate cost information. However, the more general concept potentially includes other important determiners of a task such as number of messages sent and received. By applying one or more cost measures, a task 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 switching cost measures, without any modification of the algorithm itself. This is due to the homogeneous nature of the method. Thus, the proposed algorithm is highly adaptable and can be applied within various application domains. A cost *function* is a mathematical function that quantifies or qualifies resource information to cost based on a set of cost measures. Some of the operations considered in cost modeling and analysis are cost extraction, cost generation, cost aggregation, cost evaluation, and cost analysis [4].

TABLE 3.1
An example of cost measures and cost functions

Cost measure	Cost function	Decision-making criteria
I/O connectivity	$\|X_{model}\| * \|Y_{model}\|$	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	$\|\Gamma_{model}\|$	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.
I/O and System Complexity	$\|X_{model}\| * \|Y_{model}\| * \|\Gamma_{model}\|$	The cost of a system can be captured more appropriately by considering both I/O interfaces and system complexity.
System Activity	$\|\Delta_{model}\|$	The cost of a system can be captured more appropriately by considering dynamic system behaviors.

X_{model}: a set of input interfaces, Y_{model}: a set of output interfaces,
Γ_{model}: a set of internal states, Δ_{model}: a set of internal transitions for a certain period of time
$|E|$ is a counting operator that returns the total number of elements in the given set E

A *cost tree* is a homomorphic representation of a decomposable, modular, and multi-scale task from the perspective of cost modeling and analysis. A node in the tree is classified as either *atomic* or *coupled*. An atomic node is a terminal node containing no child nodes. A coupled node is a non-terminal node holding at least one child node. A set of decomposable multi-scale tasks are easily translated to a cost tree by applying cost functions with appropriated cost measures. Each node contains a cost (or, a task and cost pair). The cost of an atomic node is generally equal to the cost of the model with which it is associated. However, the cost of a coupled node is the aggregated cost of that node and all descendants that can be reached through the tree hierarchy. Thus, the cost of a sub-tree starting from a particular node is acquired by simply retrieving the cost of the node without further expansion or exploration of the tree. So doing considerably reduces the amount of time and space required for parsing all descendants of the node and aggregating their costs during the cost evaluation process. It enables transforming NP-hard multi-scale partitioning problems to polynomial alternatives [45–47]. We show the GMP algorithm based on the method runs in polynomial time in section five.

Let D be a finite discrete set of models, $D = \{d_i | i \in \mathbb{N}\}$ where $\mathbb{N}$ is a set of positive integers and d_i is a model i. The set D refers to a decomposable set if there exists a set E such that (i) $E \subseteq D \wedge E \neq \varnothing$ and (ii) $\forall e_i \in E$, e_i is a decomposable model, $e_i = \bigcup_{j=1\ldots n} e_{ij}$. A cost tree T is a tree structure representing d_i by a_i with the preservation of structural properties of the model, for all $d_i \in D$ as shown in Figure 3.1. If the given D is populated with more than one model, $\{d_i, \ldots, d_n\}$, we repopulate D with a new virtual coupled model d_0, $D = \{d_0\}$, where $d_0 = \bigcup_{i=1\ldots n} d_i$. Every d_i in D is translated to a cost node a_i by a cost evaluation function, $f_{eval} : D \rightarrow A, d_i \in D$ and $a_i \in A$. A is the set of cost nodes representing T. If $d_i \in E$, it is also legitimate to alternatively use a cost aggregation function, $f_{aggr} : E \rightarrow A$. We can build distinct collections of cost trees by applying different aggregation methods (e.g., *summation, max*, and *average*) to d_0. So doing enables delineating a multi-scale model from various different perspectives based on aggregated cost. During the tree construction process, E shrinks when a model e_i is removed from E and D grows when e_i is expanded and its components are added to D. The process terminates when E becomes empty.

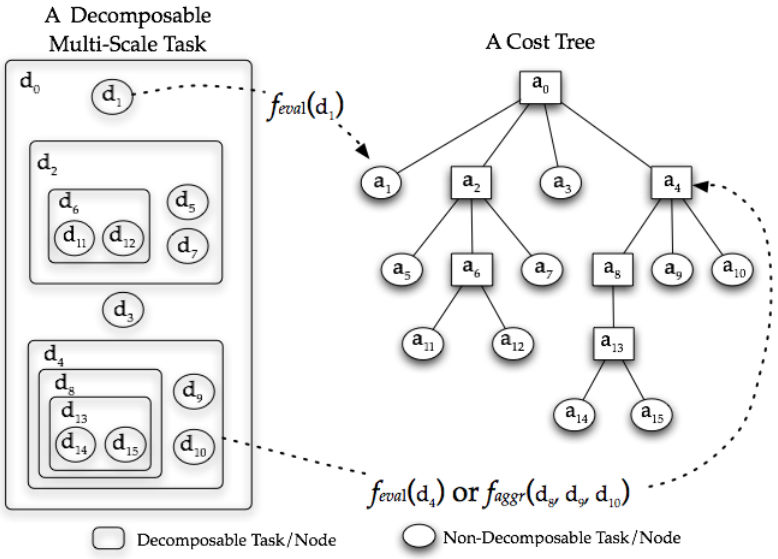


FIGURE 3.1. *Cost tree construction with a cost evaluation function and/or a cost aggregation function: A cost tree T is constructed from a decomposable task D. The cost a_i in T is computed by a cost evaluation function $f_{eval} : D \rightarrow A$. If $d_i \in E$, the cost a_i also be computed by a cost aggregation function $f_{aggr} : E \rightarrow A$, instead of f_{eval}.*

4. Generic Model Partitioning (GMP). A GMP algorithm decomposes a given multi-scale model (e.g., a coupled model in DEVS) into a set of partition blocks. It decomposes a set of models only if model decomposition produces a better partitioning result. With the minimization of model decomposition, the GMP algorithm becomes less sensitive to the depth or the complexity of the models. Minimization makes the algorithm more flexible and scalable than are partitioning algorithms based on full decomposition. A unique feature of the GMP algorithm is its support of incremental QoP improvement during the partitioning process. This property guarantees that partitioning outcomes will evolve into better alternatives without any degradation of QoP until a best partitioning result is attained. Incremental improvement enables the GMP algorithm to produce a high degree of QoP for the given model. The GMP algorithm divides into two sub-algorithms: *Initial Partitioning* and *Evaluation-Expansion-Selection*(E^2S) *Partitioning*.

4.1. Initial partitioning. The initial partitioning algorithm constructs P partition blocks from a cost tree T. Each partition block contains at least one node. The algorithm consists of four phases: *initialization, expansion, fill,* and *distribution*, as shown in Algorithm 1. All necessary data structures are created with initial values in the initialization phase (Lines 3–4). *clist* is the list containing cost nodes. It grows and shrinks, respectively, when a node is expanded from T and is assigned to a partition block. Initially, *clist* is populated with child nodes of a root node and every partition block is empty. If $P > |clist|$, at least one node expansion occurs until $|clist|$ becomes equal to or larger than P (Lines 6–12). Node expansion is a sequence of (i) identifying and removing $n_{highest}$, which is the node having the highest cost in *clist*, (ii) expanding it, (iii) and restoring its child nodes back to *clist*. If $P \leq |clist|$, select $n_{highest}$ and assign it to an empty partition block until there exist no empty partition

blocks (Lines 14–16). Finally, remaining nodes in *clist* are distributed to non-empty partition blocks until $|clist|$ becomes zero (Lines 18–20). Instead of $n_{highest}$, the node having the lowest cost in *clist*, n_{lowest} is used in the distribution phase. Initial partitioning minimizes cost disparity between partition blocks by assigning a node to each empty block in a *descending* order and distributing remaining nodes to partition blocks in an *ascending* order. Initial partitioning results of the cost tree in Figure 3.1 over various P are provided in Figure 4.1.

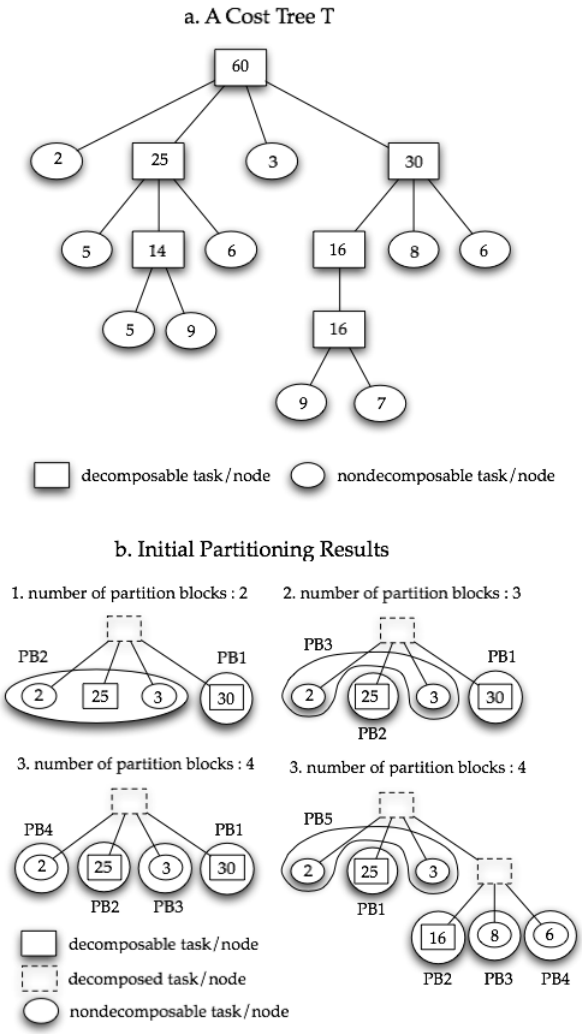


FIGURE 4.1. *Initial partitioning results of the cost tree T over various P: $f_{eval}(d_i)$ = system activity of d_i, $f_{aggr}(d_i) = \sum_j f_{eval}(d_{ij}), d_{ij} \in d_i$, and $2 \leq P \leq 5$.*

Algorithm 1 Initial Partitioning Algorithm ($\mathbb{A}_{Init}$)

Input:
T: a cost tree, P: a number of partition blocks
Return:
parray: partitioning result
Acronym:
PB: a partition block, PB_{empty}: an empty PB, $|PB| = 0$
PB_{lowest}: a PB having the lowest cost, $PB_{highest}$: a PB having the highest cost
$Node_{lowest}$: a node having the lowest cost, $Node_{highest}$: a node having the highest cost
$Node_{coupled}$: a coupled node, $Node^{highest}_{coupled}$: a coupled node having the highest cost
Operators:
$removeFrom(node, clist)$: remove a *node* from *clist*; $node \leftarrow removeFrom(node, clist)$
$addTo(node, PB)$: add a *node* to a partition block, PB; $PB^{'} \leftarrow addTo(node, PB)$
$expand(node)$: expand a *node*; a set of child nodes of the $node \leftarrow expand(node)$

```
1: procedure PB[] INITIAL-PARTITIONING(CostTree T, int P)
2:     // PHASE 1: initialize clist and parray
3:     clist := child nodes of a root node in T                          ▷ |clist| > 0
4:     parray := PB[P] // create P empty partition blocks ▷ ∀i, |parray[i]| = 0, 1 ≤ i ≤ P
5:     // PHASE 2: expand node(s), if necessary
6:     while lengthOf(clist) < numberOf(parray) do
7:         if clist contains at least one Node_coupled then
8:             clist := clist + expand(removeFrom(Node^highest_coupled, clist))
9:         else
10:            return error("cannot expand...")      ▷ (∄ Node_coupled ∈ clist) ∨ |clist| = 0
11:        end if
12:    end while                                                  ▷ |clist| ≥ |parray|
13:    // PHASE 3: fill empty partition blocks
14:    while parray contains an PB_empty do
15:        addTo(removeFrom(Node_highest, clist), PB_empty)
16:    end while                                                  ▷ ∀i, |parray[i]| > 0
17:    // PHASE 4: distribute nodes in clist into partition blocks
18:    while clist is not empty do
19:        addTo(removeFrom(Node_lowest, clist), PB_lowest)
20:    end while                                                  ▷ |clist| = 0
21:    return parray
22: end procedure
```

4.2. Evaluation-Expansion-Selection (E^{2}S) Partitioning. The E^{2}S partitioning improves the quality of partition results until no better result is attained. The algorithm consists of six phases: *initialization*, *identification*, *expansion*, *fill*, *distribution*, and *evaluation* as shown in Algorithm 2. All necessary data structures are created with initial values in the initialization phase (Lines 3–5). *parray* and *earray* are, respectively, a set of partition blocks that contain previous and next partitioning outcomes. *epartition* is the partition block that contains the highest cost in *earray*, $PB_{highest}$. *enode* is the coupled node that has the highest cost over all other nodes in $PB_{highest}$. The initial partitioning result is assigned to *parray*. After initialization, *enode* is identified from $PB_{highest}$ (Lines 7–18). The selected *enode* is expanded and *epartition* is filled with a node if it had only *enode* (Line 20 and Lines 22–24). The remaining nodes in *clist* are distributed to non-empty partition blocks until $|clist|$ becomes zero (Lines 26–28). Finally, E^{2}S partitioning recursively performs until a best result is attained (Lines 30–34). If the new partitioning result

earray is superior to the previous result *parray*, E^{2}S partitioning continues. Otherwise, returns *parray* as the best partitioning result. *superiorTo*() is a user-provided function that compares *earray* to *parray*. An example of E^{2}S partitioning results is presented in Figure 4.2. Figure 4.2.b illustrates monotonic QoP improvement of E^{2}S partitioning result.

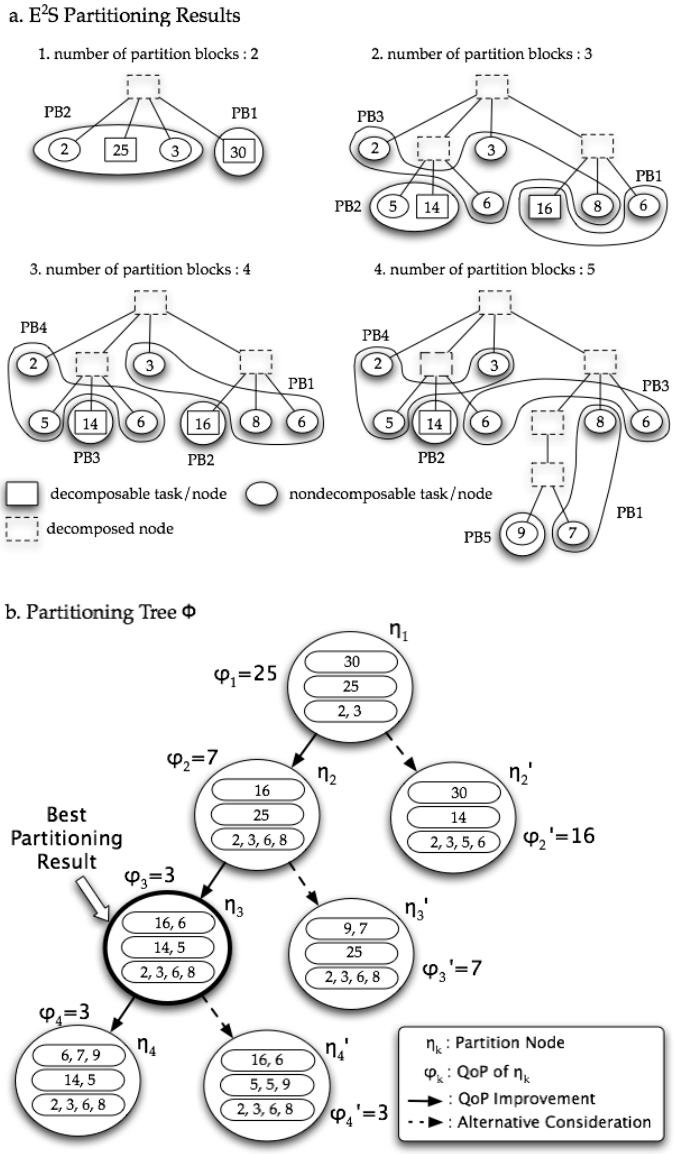


FIGURE 4.2. *E^{2}S partitioning results for various P and an example of a partitioning tree: The left figure presents partitioning results when P ranges from 2 to 5. The right figure presents a partitioning tree Φ when P is 3 and cost disparity between PB_{max} and PB_{min} is used as a partitioning quality measure, φ. $\varphi = |f_{aggr}(\mathrm{PB}_{max}) - f_{aggr}(\mathrm{PB}_{min})|$. $f_{aggr}(\mathrm{PB}_i) = \sum_j a_j$, $1 \le j \le |\mathrm{PB}_i|$ and $a_j \in \mathrm{PB}_i$. PB_{max} and PB_{min} are PB_i that respectively satisfy $f_{aggr}(\mathrm{PB}_i) \ge f_{aggr}(\mathrm{PB}_j)$ and $f_{aggr}(\mathrm{PB}_i) \le f_{aggr}(\mathrm{PB}_j), \exists i \forall j((i \ne j) \wedge (1 \le i, j \le \mathrm{P}))$. Lower φ means better QoP.*

Algorithm 2 Evaluation-Expansion-Selection (E^2S) Partitioning Algorithm ($\mathbb{A}_{E^2S}$)

Input:
parray: previous partitioning result
Return:
parray: new partitioning result
Operators:
$evaluate(PB)$: evaluate partition blocks; $value \leftarrow evaluate(PB)$
$superiorTo(PB_1, PB_2)$: check PB_1 is superior to PB_2;
$True$ or $False \leftarrow superiorTo(PB_1, PB_2)$

```
1: procedure EVALUATION-EXPANSION-SELECTION PARTITIONING(PB[] parray)
2:     // PHASE 1: initialize earray and epartition
3:     earray := parray                                          ▷ ∀i, |earray[i]| > 0
4:     epartition := PB_highest in earray                        ▷ epartition ≠ ∅
5:     enode := null                                             ▷ enode = ∅
6:     // PHASE 2: identify an expandable PB from earray
7:     while true do
8:         if epartition = null then return parray               ▷ ∀i, ∄Node_coupled ∈ earray[i]
9:         else
10:            if epartition contains Node_coupled then
11:                enode := Node_coupled^highest in epartition
12:                break                                         ▷ enode ≠ ∅
13:            else
14:                epartition := select the PB_highest from earray
15:                    excluding previously selected PBs
16:            end if
17:        end if
18:    end while                                                 ▷ ∃i, enode ∈ earray[i]
19:    // PHASE 3: expand enode and put them into clist
20:    clist := expand(removeFrom(enode, epartition))            ▷ |clist| = |enode|
21:    // PHASE 4: fill the epartition with Node_highest if epartition is empty
22:    if epartition is empty then
23:        addTo(removeFrom(Node_highest, clist), epartition)
24:    end if                                                    ▷ ∀i, |earray[i]| > 0
25:    // PHASE 5: distribute nodes to earray
26:    while clist is not empty do
27:        addTo(removeFrom(Node_lowest, clist), PB_lowest)
28:    end while                                                 ▷ |clist| = 0
29:    // PHASE 6: evaluate a new partitioning result
30:    if superiorTo(evaluate(earray), evaluate(parray)) then
31:        return Evaluation-Expansion-Selection Partitioning(earray)   ▷
               φ(earray) > φ(parray)
32:    else
33:        retrun parray                                         ▷ φ(earray) ≤ φ(parray)
34:    end if
35: end procedure
```

5. Algorithm Analysis. To make our analysis simpler, we assume a coupled model D that is translated to a cost tree $T(d, k, n)$. d is the depth of $T(d, k, n)$, k is the number of child nodes per coupled node, and n is the total number of atomic nodes. n is k^i, where $i \in [1, \ldots, d]$. The total number of nodes in $T(d, k, n)$ ranges from $\sum_{i=0}^{d-1} k^i + k$ and $\sum_{i=0}^{d} k^i$ since there exists $\sum_{i=0}^{d-1} k^i$ coupled nodes, where $d > 1$ and $k > 1$. We also assume both $f_{eval}(\cdot)$ and $f_{aggr}(\cdot)$ are $O(1)$.

5.1. Initial Partitioning Algorithm. The length of the *clist* after i node expansions l_i is

$$l_i = \begin{cases} \xi_0, & i = 0 \\ l_{i-1} + \xi_i, & i \geq 1 \end{cases} \tag{5.1}$$

where,

ξ_0 = the number of children of a root node in T

ξ_i = the number of children of the expanded node after i expansions

By substituting l_{i-1} by the sum of ξ up to i-1^{th} expansion, we can rewrite l_i as

$$l_i = l_{i-1} + \xi_i = \cdots = \sum_{j=0}^{i-1}(\xi_j - 1) - 1 + \xi_i = \sum_{j=0}^{i} \xi_j - i, \qquad i \geq 1 \tag{5.2}$$

Assume E node expansions occur to guarantee $|clist| \geq |parray|$. Then, by applying (5.2) to the *while* loop (Line 6, Algorithm 1), we can rearrange the conditional part of the loop to $(\sum_{j=0}^{E} \xi_i - E) < \mathrm{P}$. E is the total number of expansions and P is the the number of partition blocks, $|parray|$.

To make analysis simple, let ξ_i be a constant value k (i.e., k-ary tree). For a given $k \in \mathbb{N}$, the conditional part is simplied to $E < \frac{\mathrm{P}-k}{k-1}$ by substituting k for ξ_i. Since $E \in \mathbb{N}$, the total number of node expansions needed in the initial partitioning becomes $\lceil \frac{\mathrm{P}-k}{k-1} \rceil, 1 < k < \mathrm{P}$. No expansion occurs when $k \geq \mathrm{P}$. The length of *clist* after the expanding phase, l_E, is described by P and k by substituting $\lceil \frac{\mathrm{P}-k}{k-1} \rceil$ for i and k for ξ_i in (5.2). l_E is k when no expansion occurs because the *clist* initially populated with the children of the root node.

$$l_E = \begin{cases} (k\text{-}1) \cdot \left\lceil \frac{\mathrm{P}-k}{k-1} \right\rceil + k, & 1 < k < \mathrm{P} \\ k, & k \geq \mathrm{P} \end{cases} \tag{5.3}$$

P comparisons occur in the filling phase because every empty partition in the *parray* is filled with a cost node that is extracted from the *clist*. (l_E - P) comparisons occur in the distribution phase because the remaining *clist* nodes are distributed into the *parray*.

DEFINITION 5.1. For execution time complexity analysis of $\mathbb{A}_{initial}$, we define

$\xi(n)$*:* expand the node, n

δ^{init}_{part}*:* time required for executing $\mathbb{A}_{initial}$

$\delta(clist, nodes, add)$*:* time required for adding nodes to the *clist*

$\delta(clist, nodes, remove)$*:* time required for removing nodes from the *clist*

$\delta(parray, size, create)$*:* time required for creating the parray with size empty blocks

$\delta(PB_i, nodes, add)$*:* time required for adding nodes to the PB_i

Algorithm execution time is the sum of time spent in each phase of the algorithm. That is, $\delta^{init}_{part} = \delta_{init} + \delta_{expand} + \delta_{fill} + \delta_{dist}$. We can rewrite it to

$$\delta^{init}_{part} = \begin{cases} \delta_{init} + \sum_{i=1}^{E} \delta_{expand_i} + \sum_{i=1}^{\mathrm{P}} \delta_{fill_i} + \sum_{i=1}^{l_E - \mathrm{P}} \delta_{dist_i}, & 1 < k < \mathrm{P} \\ \delta_{init} + 1 + \sum_{i=1}^{\mathrm{P}} \delta_{fill_i} + \sum_{i=1}^{k-\mathrm{P}} \delta_{dist_i}, & k \geq \mathrm{P} \end{cases} \tag{5.4}$$

where,

$$\delta_{init} = \delta(clist, \xi(Node_{root}), add) + \delta(parray, P, create)$$

$$\delta_{expand_i} = \delta(clist, Node^{highest}_{coupled}, remove) + \delta(clist, \xi(Node^{highest}_{coupled}), add)$$
$$\delta_{fill_i} = \delta(clist, Node_{highest}, remove) + \delta(PB_{empty}, Node_{highest}, add)$$
$$\delta_{dist_i} = \delta(clist, Node_{lowest}, remove) + \delta(PB_{lowest}, Node_{lowest}, add)$$

By applying E and l_E to (5.4), we get

$$\delta^{init}_{part} = \begin{cases} \delta_{init} + \sum_{i=1}^{\lceil \frac{\mathrm{P}-k}{k-1} \rceil} \delta_{expand_i} + \sum_{i=1}^{\mathrm{P}} \delta_{fill_i} + \sum_{i=1}^{(k-1)\lceil \frac{\mathrm{P}-k}{k-1} \rceil + k - \mathrm{P}} \delta_{dist_i}, & 1 < k < \mathrm{P} \\ \delta_{init} + 1 + \sum_{i=1}^{\mathrm{P}} \delta_{fill_i} + \sum_{i=1}^{k-\mathrm{P}} \delta_{dist_i}, & k \geq \mathrm{P} \end{cases} \tag{5.5}$$

To make analysis of the algorithm simple, assume it takes one time unit either to run an operator or to evaluate a conditional statement. Then, δ_{init} takes 3 units: 1 unit for the expansion of the root node and 2 units for the initialization of *clist* and *parray*. δ_{expand_i} takes 5 units: 2 units for the evaluation of conditional parts of both *while* and if loops and 3 units for the execution of the *remove-expand-add* operation. Both δ_{fill_i} and δ_{dist_i} take 3 units: 1 unit for the evaluation of conditional part of the *while* loop and 2 units for the execution of the *remove-add* operation. By substituting all $\delta(\cdot)$ in (5.5) by appropriate execution time,

$$\delta^{init}_{part} = \begin{cases} 3 + \left\lceil \frac{\mathrm{P}-k}{k-1} \right\rceil \cdot 5 + \mathrm{P} \cdot 3 + \left((k-1) \left\lceil \frac{\mathrm{P}-k}{k-1} \right\rceil + k - \mathrm{P} \right) \cdot 3, & 1 < k < \mathrm{P} \\ 3 + 1 + \mathrm{P} \cdot 3 + (k - \mathrm{P}) \cdot 3, & k \geq \mathrm{P} \end{cases} \tag{5.6}$$

By rearranging (5.6), we get

$$\delta^{init}_{part} = \begin{cases} (3k+2) \left(\left\lceil \frac{\mathrm{P}-k}{k-1} \right\rceil + 1 \right) + 1, & 1 < k < \mathrm{P} \\ 3k + 4, & k \geq \mathrm{P} \end{cases} \tag{5.7}$$

(5.7) shows that, for a given partition block size P, the total execution time of the initial partitioning algorithm is more sensitive to the number of children of a coupled node rather than the size or the complexity of a given model. It also implies that the performance of the algorithm is highly bound to the number of expansions, E, rather than model complexity.

5.2. E^{2}S Partitioning Algorithm. DEFINITION 5.2. For execution time complexity analysis of $\mathbb{A}_{E^2S}$, we define

ξ_{enode}: the number of child nodes expanded from *enode*, $|\xi(enode)|$
γ: the number of recursions until a best partitioning result is attained
$\delta^{E^2S}_{part}$: time required for executing $\mathbb{A}_{E^2S}$

The E^{2}S partitioning algorithm has six phases as described in Algorithm 2. Thus, the total execution time of the algorithm is the sum of time spent in those phases: $\delta^{E^2S}_{part} = \delta_{init} + \delta_{identify} + \delta_{expand} + \delta_{fill} + \delta_{dist} + \delta_{eval}$. In the *while* loop, l comparisons occur to find the *enode* in the identification phase, $1 \leq l \leq \mathrm{P}$. At most, ξ_{enode} comparisons occur in the distribution phase to redistribute all nodes of the *clist* to the *earray*. No comparisons occur in other phases.

$$\delta^{E^2S}_{part} = \delta_{init} + \sum_{i=1}^{l} \delta_{identify_i} + \delta_{expand} + \varepsilon \cdot \delta_{fill} + \sum_{i=1}^{\xi_{enode}-\varepsilon} \delta_{dist_i} + \delta_{eval} \tag{5.8}$$

where,

l: the total number of comparisons occurred in the while loop to find *enode*,
ε: 1 if *epartition* is empty after the expansion phase. Otherwise, 0

Since l comparisons in the identification phase divide into l-1 for partition blocks having no coupled node and 1 for the partition block having at least one coupled node, we revise (5.8) to

$$\delta_{init}+\sum_{i=1}^{l-1}\delta_{identify,\neg enode}+\delta_{identify,enode}+\delta_{expand}+\varepsilon\cdot\delta_{fill}+\sum_{i=1}^{\xi_{enode}-\varepsilon}\delta_{dist_i}+\delta_{eval} \quad (5.9)$$

where,

l: the total number of comparisons occurred in the while loop to find *enode*,

ε: 1 if *epartition* is empty after the expansion phase. Otherwise, 0

$\delta_{identify,\neg enode}$: time need for handling *epartition* having no coupled node

$\delta_{identify,enode}$: time need for handling *epartition* having at least one coupled node

To simplify the analysis, assume it takes one time unit to run an operator or to evaluate a conditional statement. Then, δ_{init} takes 3 units to initialize *earray*, *epartition*, and *enode*. $\delta_{identify,\neg enode}$ takes 4 units to handle *epartition* having no coupled node. $\delta_{identify,enode}$ takes 5 units to identify *enode*. δ_{expand}, δ_{fill}, and δ_{dist_i} take 3 units, respectively. δ_{eval} takes 3 + $\delta_{part}^{'E2S}$ units. $\delta_{part}^{'E^2S}$ is equal to the time need for running the algorithm recursively until the best result is attained. $\delta_{part}^{'E^2S}$ is 1 if *evaluate*(*earray*) is not better than *evaluate*(*parray*). By substituting all $\delta(\cdot)$ in (5.9) by appropriate execution time, we get

$$\delta_{part}^{E^2S}=3+(l-1)\cdot 4+5+3+\xi_{enode}\cdot 3+3+\delta_{part}^{'E^2S}=4l+3\xi_{enode}+10+\delta_{part}^{'E^2S} \quad (5.10)$$

Assume $\delta_{part}^{'E^2S}$ runs γ times recursively to attain a best result. Then, we rewrite (5.10) to

$$\delta_{part}^{E^2S}=4l+3\xi_{enode}+10+\sum_{i=1}^{\gamma}\left(4l_i+3\xi_{enode}^{i}+10\right)+1 \quad (5.11)$$

where,

l_i: l at i-th recursion in $\delta_{part}^{'E^2S}$

ξ_{enode}^{i}: ξ_{enode} at i-th recursion in $\delta_{part}^{'E^2S}$

In most cases, l_i is 1. However, it could vary dynamically depending on the content of the *parray*. We approximate l_i by introducing $\bar{l}$, the average of l_i as follows

$$\delta_{part}^{E^2S}=\sum_{i=0}^{\gamma}\left(4\bar{l}+3\xi_{enode}^{i}+10\right)+1 \quad (5.12)$$

where,

$\bar{l}$: the average of l, $\frac{1}{\gamma+1}\cdot\sum_{i=0}^{\gamma}l_i$

ξ_{enode}^{i}: ξ_{enode} at i-th recursion

For a k-ary cost tree, we rewrite (5.12) by substituting ξ_{enide}^{i} by k as follows

$$\delta_{part}^{E^2S}=\sum_{i=0}^{\gamma}\left(4\bar{l}+3k+10\right)+1 \quad (5.13)$$

(5.13) implies that the total execution time of the E^{2}S partitioning algorithm is sensitive to the degree of QoP rather than the size or the complexity of the given hierarchical model.

5.3. Worst Case Analysis.

5.3.1. Initial Partitioning Algorithm. For a worst case in the initial partitioning, assume the cost tree $T(d, k, n)$ sustains the following constraints.

i. The total number of atomic nodes n is k^d,
ii. The number of partition blocks P is k^d,
iii. $d \gg$ k

E is induced to $\frac{k^d-1}{k-1}$ from the constraints i and ii. The constraints imply that there exist $\sum_{i=0}^{d-1} k$ coupled nodes and they are all expanded. l_E is computed to $k^d + k - 1$ by substituting $\frac{k^d-1}{k-1}$ for i and k for ξ_i (5.2). By substituting all $\delta(\cdot)$ in (5.4) by appropriate execution time and rearranging it, we get

$$\delta_{part}^{init} = 3k^d + 5\,\frac{k^d-1}{k-1} + 3k = 3n + 5\,\frac{n-1}{n^{1/d}-1} + 3n^{1/d} \tag{5.14}$$

Since k is $n^{1/d}$ and $n \gg d$, $O(\delta_{part}^{init})$ is $O(n)$ for $T(d, k, n)$, $d \gg k > 1$, $n = k^d$.

5.3.2. E^{2}S Partitioning Algorithm. For the worst case in the E^{2}S partitioning, assume an additional constraint

iv. A new partitioning result is always superior to the previous one.

In the case, l_i is 1 because the $\mathrm{PB}_{highest}$ always contains at least one couple node. We can compute γ for the given $T(d, k, n)$ by

$$\gamma = \sum_{i=1}^{d-1} k^i = \frac{k^d-1}{k-1} - 1,\ k > 1 \tag{5.15}$$

By substituting l_i and γ with 1 and $\frac{k^d-1}{k-1}$-1, respectively, in (5.13), we get

$$\delta_{part}^{E^2S} = \left(\frac{k^d-1}{k-1} - 1 + 1\right) \cdot (4 + 3k + 10) + 1 \tag{5.16}$$

We rewrite (5.16) in terms of n.

$$\delta_{part}^{E^2S} = 3 \cdot \left(\frac{n-1}{n^{1/d}-1}\right) \cdot \left(n^{1/d} + \frac{14}{3}\right) + 1 \tag{5.17}$$

$O(\delta_{part}^{E^2S})$ is $O(n)$ for the cost tree $T(d, k, n)$, $d \gg k > 1$, $n = k^d$.

5.4. Parameter Optimization.

5.4.1. Optimal Number of Partition Blocks. For a given coupled model, the optimal number of partition blocks P_{opt} is identified by searching P that produces minimum execution time from the following equation. $\delta_p^K(\mathbb{A}_{Init})$ is the execution time of the initial partitioning algorithm for a fixed K and an aribtrary p.

$$\min_{1<p\le\infty}\{\delta_p^K(\mathbb{A}_{Init})\} = \begin{cases} \min_{K<p\le\infty}\{(3K+2)\left(\left\lceil\frac{p-K}{K-1}\right\rceil + 1\right) + 1\}, & K < p \le \infty \\ 3K+4, & p \le K \end{cases} \tag{5.18}$$

By rearranging (5.18), we get

$$\min\{\delta_p^K(\mathbb{A}_{Init})\} = \begin{cases} \min_{K<p\le\infty}\{\frac{3K+2}{K-1}\,p + \frac{3K+2}{K-1} + 1\}, & K < p \le \infty \\ 3K+4, & p \le K \end{cases} \tag{5.19}$$

P_{opt} is $K+1$ when $K < p \leq \infty$ because $\delta_p^K(\mathbb{A}_{Init})$ grows linearly as p increases. $\delta_{K+1}^K(\mathbb{A}_{Init})$ produces minimum execution time. P_{opt} is p when $p \leq K$ because $\delta_p^K(\mathbb{A}_{Init})$ is governed by only K independent from p.

$$P_{opt} = \begin{cases} K+1, & K < p \leq \infty \\ p, & p \leq K \end{cases} \tag{5.20}$$

In $\mathbb{A}_{E^2S}$, it is unnecessary to compute P_{opt}. P is set by $\mathbb{A}_{Init}$ and is indirectly referenced by $\mathbb{A}_{E^2S}$ through the set of partition blocks. P is an implicit and non-permutable value in $\mathbb{A}_{E^2S}$. Also, $\delta_{part}^{E^2S}$ is mainly bound to γ rather than P.

5.4.2. Optimal Cost Tree for a Particular Number of Partition Blocks. For a fixed number of partition blocks P, the optimal number of k-ary cost tree k_{opt} is identified by searching k that produces minimum execution time from the following equation. $\delta_P^k(\mathbb{A}_{Init})$ is the execution time of the initial partitioning algorithm for a fixed P and an arbitrary k.

$$\min_{1<k\leq\infty}\{\delta_{\mathrm{P}}^k(\mathbb{A}_{Init})\} = \begin{cases} \min_{1<k<P}\{(3k+2)\left(\left\lceil\frac{P-k}{k-1}\right\rceil+1\right)+1\}, & 1<k<P \\ 3k+4, & k \geq P \end{cases} \tag{5.21}$$

By rearranging (5.21), we get

$$\min_{1<k\leq\infty}\{\delta_{\mathrm{P}}^k(\mathbb{A}_{Init})\} = \begin{cases} \min_{1<k<P}\{5(\mathrm{P}-1)\frac{1}{k-1}+3(\mathrm{P}-1)+1\}, & 1<k<P \\ 3k+4, & k \geq P \end{cases} \tag{5.22}$$

k_{opt} is P - 1 when $1 < k <$ P because $\delta_P^k(\mathbb{A}_{Init})$ decreases linearly as k increases. $\delta_{\mathrm{P}}^{\mathrm{P}-1}(\mathbb{A}_{Init})$ produces minimum execution time. k_{opt} is P when $k \geq$ P because $\delta_P^k(\mathbb{A}_{Init})$ increases linearly as k increase.

$$k_{opt} = \begin{cases} \mathrm{P}-1, & 1<k<P \\ \mathrm{P}, & k \geq P \end{cases} \tag{5.23}$$

In $\mathbb{A}_{E^2S}$, it is not necessary to compute k_{opt}. Once a model is selected by $\mathbb{A}_{Init}$, $\mathbb{A}_{E^2S}$ cannot dynamically permutate structural properties of the model.

6. Experimental Results. A series of experiments have been conducted to evaluate the GMP algorithm and compare it to two multi-scale partitioning algorithms, *random* and *ratio-cut*. A set of quality and performance measures has been applied to the results. All experiments were performed on a small-scale Beowulf cluster system which consists of one root node, seven compute nodes, a Gigabit switch, and a dedicated Gigabit network. Each node is equipped with a single Intel Pentium 4 3 Ghz CPU, 2GB PC 3200 DDR 400 SDRAM memory, a 80 GB 7200 RPM hard disk, and a Gigabit Ethernet card. The root node has an extra Gigabit Ethernet card to access public Internet.

A multi-scale decomposable DEVS coupled model is generated with a particular cost pattern listed in Table 6.1. Six cost patterns are used to represent a wide range of computational workload of the model: $C_{unitstep}$ and C_{exp} for low workload, C_{pareto} and C_{invgau} for medium workload, and $C_{uniform}$ and $C_{lognormal}$ for high workload. The generated coupled model is partitioned into a set of partition blocks by each algorithm. The blocks are dispatched to a set of processors and decomposed models

TABLE 6.1
Cost patterns for generating various computational workload distributions of a model [48–51]

Pattern	PMF	Parameters	Distribution	Load
$C_{unitstep}$	$\delta(x)$	None	Unit Step	Low
C_{exp}	$\lambda e^{-\lambda x}$	$\lambda = 0.05$	Exponential	
C_{pareto}	$\alpha k^{\alpha} x^{-\alpha-1}$	$\alpha = 1.245$, $k = 3$	Pareto	Medium
$C_{invgaus}$	$\sqrt{\frac{\lambda}{2\pi x^3}} e^{\frac{-\lambda(x-\mu)^2}{2\mu^2 x}}$	$\mu = 3.86$, $\lambda = 9.46$	Inverse Gaussian	
$C_{uniform}$	1	None	Uniform	High
$C_{lognorm}$	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(lnx-\mu)^2}{2\sigma^2}}$	$\mu = 5.929$, $\sigma = 0.321$	Lognormal	

PMF: Probability Mass Function , $\delta(x)$: a PMF that returns 1 when $x = a$. otherwise, returns 0.

in the blocks are executed in parallel. Quality and performance of algorithms are computed by applying qualitative and quantitative measures to the blocks.

A set of measures used in our experiments is listed in Table 6.2. The cost of a partition block is computed by four cost measures: ϕ_{norm}, ϕ_{diff}, ϕ_{dist}, and ϕ_{var}. The quality of a set of partition blocks is evaluated by two QoP measures: $\varphi_{avg\text{-}diff}$ and $\varphi_{avg\text{-}diff}$. QoP evolution is traced by profiling the quality of the set until the best partitioning result is attained. Execution time of the set is collected by four time measures: δ_{total}, δ_{accum}, δ_{avg}, and, δ_{sqrt}.

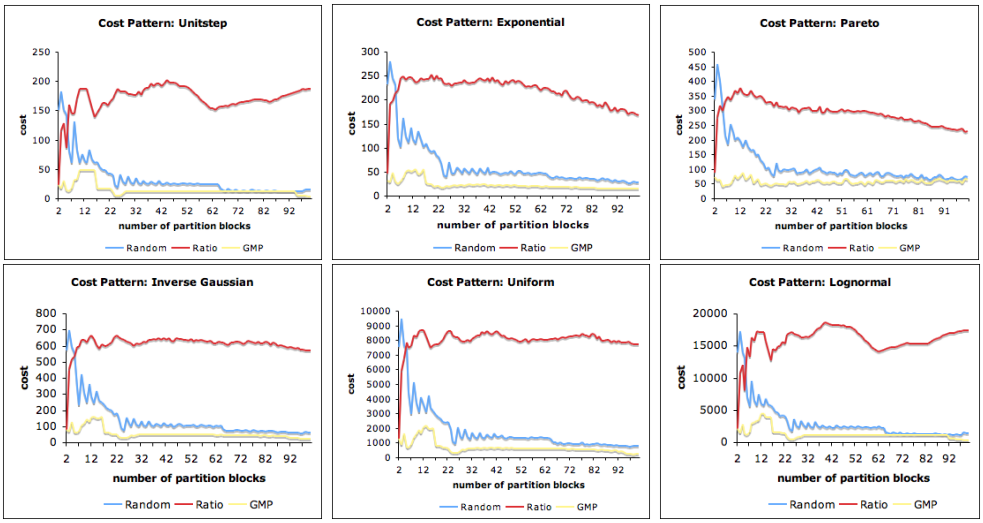
TABLE 6.2
A set of measures for quality and performance evaluation

Measure	Mathematical representation	Description
ϕ_{norm}	$f_{aggr}(\mathrm{PB}_i)/\max\{f_{aggr}(\mathrm{PB}_j)\}_{j=1}^{\mathrm{P}}$	normalized cost
ϕ_{diff}	$\sum_{j=1}^{\mathrm{P}} \lvert f_{aggr}(\mathrm{PB}_i) - f_{aggr}(\mathrm{PB}_j)\rvert$	cost difference
ϕ_{dist}	$\sum_{j=1}^{\mathrm{P}} \lvert f_{aggr}(\mathrm{PB}_i) - f_{aggr}(\mathrm{PB}_j)\rvert$	cost distance
ϕ_{var}	$\sqrt{\lvert f_{aggr}(\mathrm{PB}_i) - \sum_{j=1}^{\mathrm{P}} f_{aggr}(PB_j)/\mathrm{P}\rvert}$	cost variance
$\varphi_{min\text{-}max}$	$\lvert max\{f_{aggr}(\mathrm{PB}_i)\}_{i=1}^{\mathrm{P}} - min\{f_{aggr}(\mathrm{PB}_i)\}_{i=1}^{\mathrm{P}}\rvert$	min-max disparity
$\varphi_{avg\text{-}diff}$	$\sum_{i=1}^{\mathrm{P}}\sum_{j=1}^{\mathrm{P}} \lvert f_{aggr}(\mathrm{PB}_i) - f_{aggr}(\mathrm{PB}_j)\rvert/\mathrm{P}$	average difference
δ_{total}	$\max\{\tau(\mathrm{PB}_i)\}_{i=1}^{\mathrm{P}}$	total execution time
δ_{accum}	$\sum_{i=1}^{\mathrm{P}} \tau(\mathrm{PB}_i)$	accumulated execution time
δ_{avg}	$\sum_{i=1}^{\mathrm{P}} \tau(\mathrm{PB}_i)/\mathrm{P}$	average execution time
δ_{sqrt}	$\sqrt{\sum_{i=1}^{\mathrm{P}} \tau(\mathrm{PB}_i)/\mathrm{P}}$	square root execution time

$f_{aggr}(\mathrm{PB}_i) = \sum c(t_j)$, where $c(t_j)$ is the cost of a model $t_j \in \mathrm{PB}_i$, $\tau(\mathrm{PB}_i)$ = time need to execute all models in PB_i

Figure 6.1 represents QoP experimental results of $T(7, 4, 400)$. Two QoP measures $\varphi_{min-max}$ and $\varphi_{avg-diff}$ are used to evaluate the quality of the partitioning results produced by each algorithm. For a particular number of partition blocks P, a cost pattern $C_{pattern}$, and a QoP measure, each algorithm is applied to 20 different computational workloads. The average of the 20 experiments is computed and illustrated as a single point in the figure. We measured QoP of partitioning results by varying P from 2 to 100. The GMP algorithm produces superior QoP outcomes compared to other algorithms for both QoP measures. It is mainly because the GMP algorithm minimizes the cost disparities between partition blocks with minimum model decomposition. However, QoP outcomes of other algorithms were highly sensitive to P, $C_{pattern}$, and QoP measures. The QoP experimental results shown in Figure 6.1.

a. QoP measure: min-max disparity



b. QoP measure: average difference

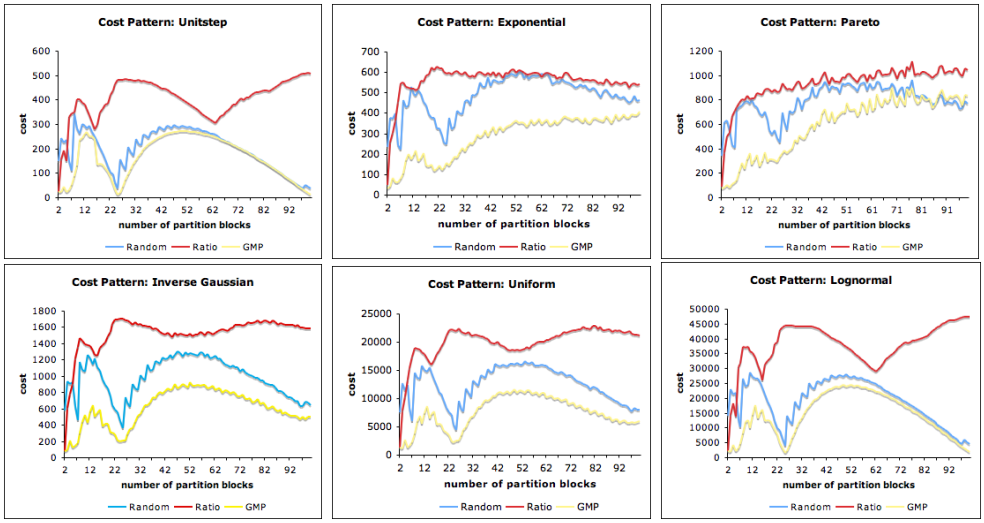


FIGURE 6.1. *QoP evaluation of partitioning results over various cost patterns and numbers of partition blocks: A point in each figure represents the average of 20 different executions with respect to a set of a cost pattern, a number of partition blocks, and a partitioning algorithm,* $C_{pattern}$, P, $\mathbb{A}_{partition}$. $C_{pattern} \in \{C_{unitstep}, C_{exp}, C_{pareto}, C_{invgaus}, C_{uniform}, C_{lognorm}\}$. $2 \leq P \leq 100$. $\mathbb{A}_{partition} \in \{Random, Ratio\text{-}Cut, GMP\}$. *Two QoP measures,* $\varphi_{min-max}$ *and* $\varphi_{avg-diff}$, *are applied to* $T(7, 4, 400)$. *The lower value in the Y axis represents the better result.*

All experimental results in the figure are summarized in Table 6.3.

TABLE 6.3
The summary of QoP experiments of $T(7, 4, 400)$

	$\varphi_{min-max}$			$\varphi_{avg-diff}$		
Cost Pattern	Random	Ratio-Cut	GMP	Random	Ratio-Cut	GMP
$C_{unitstep}(x)$	33.44	170.28	14.77	198.79	402.94	165.11
$C_{exp}(x)$	62.18	218.27	22.57	484.83	559.83	284.50
$C_{pareto}(x)$	110.44	292.09	56.84	788.20	929.16	595.32
$C_{invgaus}(x)$	139.83	609.87	53.60	984.37	1520.27	603.98
$C_{uniform}(x)$	1799.33	8000.09	689.31	12492.31	19945.52	7521.78
$C_{lognorm}(x)$	3077.86	15873.68	1303.46	18884.30	37774.36	14240.95
Average	870.51	4194.05	356.76	5638.80	10188.68	3901.94

unit: second

Figure 6.2 represents execution time measurement results of $T(7, 4, 50)$. Two time measures δ_{accum} and δ_{avg} are used to evaluate performance of the partitioning results produced by each algorithm. Model execution time is measured for the case of only one partition block allocation per processor. For a particular number of processors np and $C_{pattern}$, and a time measure, each algorithm is applied to 5 different computational workloads. In most experiments, the GMP algorithm requires the shortest execution time. The execution time measurement results are shown in Figure 6.2. All experimental results in the figure are summarized in Table 6.4.

TABLE 6.4
The summary of execution time measurement of $T(7, 4, 50)$

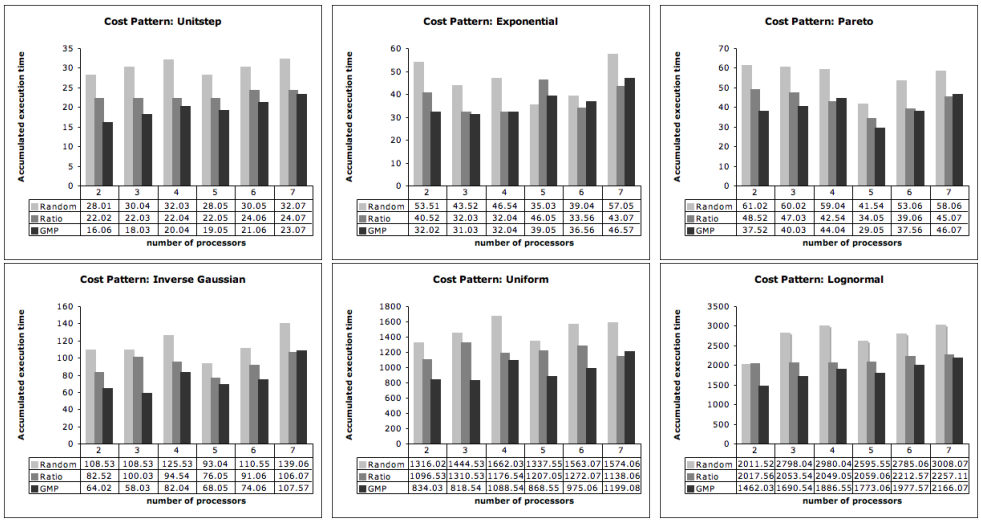
	δ_{accum}			δ_{avg}		
Cost Pattern	Random	Ratio-Cut	GMP	Random	Ratio-Cut	GMP
$C_{unitstep}(x)$	30.04	22.71	19.55	7.87	5.95	4.94
$C_{exp}(x)$	45.78	37.88	36.21	12.43	9.98	9.15
$C_{pareto}(x)$	55.45	42.71	39.04	15.12	11.72	10.29
$C_{invgaus}(x)$	114.21	91.71	75.63	29.79	23.96	18.86
$C_{uniform}(x)$	1482.88	1200.13	963.97	384.65	315.87	244.92
$C_{lognorm}(x)$	2696.38	2108.15	1825.97	682.74	551.43	459.97
Average	737.46	583.88	493.39	188.77	153.15	124.69

unit: second

7. Summary and Conclusions. This paper presents a new GMP algorithm. It efficiently decomposes a multi-scale model into a set of partition blocks using the cost modeling and analysis method. It also produces monotonically improved partitioning results with minimum model decomposition. The method enables abstracting subjective, heterogeneous, domain-dependent information into objective, homogeneous, domain-independent cost information. With the selection of different methods of cost measures, cost evaluation, and cost aggregation, the proposed algorithm performs various partitioning strategies without any modification of the generic partitioning logics. Because each cost measure is a parametric method, and partitioning logic is described over the homogeneous cost space, the algorithm is generic and applicable to any family of models provided there is a way to manipulate the appropriate cost information.

Algorithm analysis and experimental results show that the GMP algorithm is efficient and produces high quality partitioning results. The algorithm execution time is $O(n)$ in the worst case scenario. The experimental results show that the GMP algorithm produces partitioning results that are superior to those from other algorithms.

a. Execution time measure: accumulated execution time



b. Execution time measure: average execution time

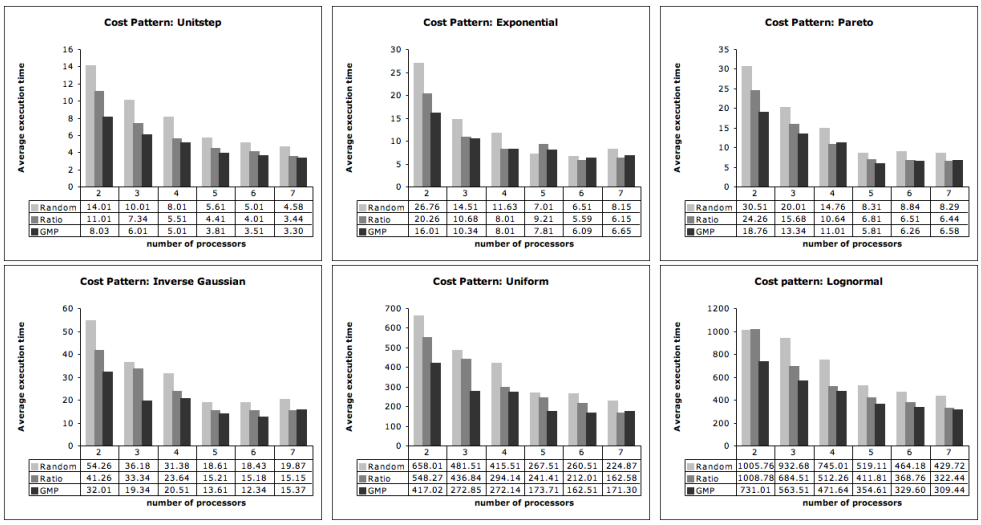


FIGURE 6.2. *Model execution time measurement of partitioning results over various cost patterns and numbers of processors: Each mark in the figure is the average of 5 different executions with respect to a set of a cost pattern, a number of processors, and a partitioning algorithm,* $C_{pattern}$, np, $\mathbb{A}_{partition}$. $C_{pattern} \in \{C_{unitstep}, C_{exp}, C_{pareto}, C_{invgaus}, C_{uniform}, C_{lognorm}\}$. $2 \leq np \leq 7$. $\mathbb{A}_{partition} \in \{Random, Ratio - Cut, GMP\}$. *Two execution time measures ,* δ_{accum} *and* δ_{avg}, *are applied to* $T(7, 4, 50)$. *The lower value in the Y axis represents the better result.*

A set of GMP-based multi-scale model partitioners has been implemented over distributed network middleware to support large-scale discrete-event oriented simulations. Because the algorithm is generic, concise, and reconfigurable, it can easily evolve to accommodate static and dynamic resource management system components that efficiently handle multi-scale models in large-scale distributed and parallel simulation systems.

The pace of M&S driven systems biology research using constructive, multi-scale

models is expected to increase. However, for efficient execution of these models, we need generic but domain-aware, multi-scale partitioning algorithms. The GMP algorithm meets the requirement and has been successfully implemented as a part of multi-scale model partitioners in various large-scale distributed simulation frameworks. A wide range of distinctive multi-scale, constructive, modular biological system models can be easily managed by changing or revising cost functions without any modification of generic partitioning programming logics. That ability positions the GMP algorithm to be effective in large-scale M&S driven systems biology research.

Acknowledgments. The authors would thank Dr. James Nutaro, Dr. Hessam Sarjoughian, and anonymous reviewers for their constructive comments and suggestions that helped improve the content of the paper.

REFERENCES

[1] H. Kitano, *Computational systems biology*, Nature, 420 (2002), pp. 206–210.
[2] T. Idekeer, T. Galitski, and L. Hood, *A new approach to decoding life: Systems biology*, Annual Review of Genomics and Human Genetics, 2 (2001), pp. 343–372.
[3] L. Hood, *Systems biology: Integrating technology, biology, and computation*, Mechanisms of Ageing and Development, 124 (2003), pp. 9–16.
[4] S. Park, *Cost-based Partitioning for Distributed Simulation of Hierarchical, Modular DEVS Models*, PhD thesis, Department of Electrical and Computer Engineering, University of Arizona, 2003.
[5] S. Park, C. A. Hunt, and G. E. P. Ropella, *PISL: A large-scale in silico expermentation framework for agent-directed physiological models*, in The Proceeding of the 2005 Agent-Directed Simulation Symposium, Spring Simulation Conference, San Diego, CA, April 2005.
[6] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*, Academic Press, San Deigo, CA, 2nd ed., 2000.
[7] H. L. Vangheluwe, *DEVS as a common denominator for multi-formalism hybrid systems modeling*, in Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control System Design, Anchorage, Alaska, 2000.
[8] B. P. Zeigler, *DEVS today: Recent advances in discrete event-based information technology*, in 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, Orlando, Florida, 2003, pp. 141–161.
[9] R. Djafarzadeh, G. Waineer, and T. Mussivand, *DEVS modeling and simulation of the cellular metabolism by mitochondria*, in The Proceeding of the 2005 DEVS Integrative M&S Symposium, Spring Simulation Conference, San Diego, California, 2005, pp. 55–62.
[10] X. Hu and D. H. Edwards, *Behaviorsim: A simulation environment to study animal behavioral choice mechanisms*, in The Proceeding of the 2005 DEVS Integrative M&S Symposium, Spring Simulation Conference, no. 29 - 35, San Diego, California, 2005.
[11] S. Biermann, A. M. Uhrmacher, and H. Schumann, *Supporting multi-level models in systems biology by visual methods*, in Proceedings of the 18th European Simulation Multiconference, Magdeburg, Germany, June 2004.
[12] A. M. Uhrmacher, D. Degenring, and B. P. Zeigler, *Discrete event multi-level models for systems biology*, Lecture Notes in Computer Science, 3380 (2005), pp. 66 – 89.
[13] C. Seo, S. Park, B. Kim, S. Cheon, and B. P. Zeigler, *Implementation of distributed high-performance devs simulation framework in the grid computing environment*, in 2004 Advanced Simulation Technologies Conference (ASTC '04) - High Performance Computing Symposium 2004 (HPCS 2004), Arlington, VA, USA, 2004.
[14] S. Cheon, C. Seo, S. Park, and B. P. Zeigler, *Design and implementation of distributed DEVS simulation in a peer-to-peer network system*, in 2004 Advanced Simulation Technologies Conference (ACTS '04) - Design, Analysis, and Simulation of Distributed Systems Symposium 2004 (DASD 2004), Arlington, VA, USA, 2004.
[15] A. Pothen, *Graph partitioning algorithms with applications to scientific computing*, Parallel Numerical Algorithms, (1996), pp. 323–368.
[16] P.-O. Fjällström, *Algorithms for graph partitioning: A survey*, Linkköping Electronic Articles in Computer and Information Science, 3 (1998).

[17] C. J. Alpert and A. B. Kahng, *Recent directions in netlist partitioning: a survey*, SIAM Journal on Scientific Computing, 16 (1995), pp. 452–469.
[18] K. Schloegel, G. Karypis, and V. Kumar, *Graph partitioning for high performance scientific simulations*, Computing Reviews, 45 (2004).
[19] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing, 20 (1998), pp. 359–392.
[20] S. T. Barnard and H. D. Simon, *A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems*, Concurrency: Practice & Experience, 6 (1994), pp. 101–117.
[21] Y. G. Saab, *An effective multilevel algorithms for bisecting graphs and hypergraphs*, IEEE Transactions on Computers, 53 (2004), pp. 641–652.
[22] M. O. Möller and R. Alur, *Heuristics for hierarchical partitioning with application to model checking*, in Correct Hardware Design and Verification Methods: 11th IFIP WG 10.5 Adanced Research Working Conference, Livingston, Scotland, UK, 2001, pp. 71–85.
[23] C. Walshaw, *Multilevel refinement for combinatorial optimisation problems*, Annals of Operations Research, 131 (2004), pp. 325–372.
[24] K. Kim, T. Kim, and K. Park, *Hierarchical partitioning algorithm for optimistic distributed simulation of devs models*, Journal of Systems Architecture, 44 (1998), pp. 433–455.
[25] G. Zhang and B. P. Zeigler, *Mapping hierarchical discrete models to multiprocessor system: Concept, algorithm, and simulation*, Journal of Parallel and Distributed Computing, 9 (1990), pp. 271–280.
[26] B. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, The Bell Technical Journal, 49 (1970), pp. 291–307.
[27] S. Dutt, *New faster kernighan-lin-type graph-partitioning algorithms*, in Proceedings of the 1993 IEEE International conference of Computer-aided design, 1983, pp. 370–377.
[28] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in IEEE-ACM 19th Design Automation Conference, Las Vegas, NV, 1982, pp. 175–181.
[29] R. Diekmann, B. Monien, and R. Preis, *Using helpful sets to improve graph bisections*, Tech. Rep. TR-RF-94-008, Dept. of Computer Science, University of Paderborn, Germany, 1994.
[30] J. R. Gilbert and E. Zmijewski, *A parallel graph partitioning algorithm for a message-passing multiprocessor*, International Journal of Parallel Programming, 16 (1987), pp. 427 – 449.
[31] M. J. Berger and B. H. Bokhari, *A partitioning strategy for non-uniform problems across multiprocessors*, IEEE Transactions on Computers, 36 (1987), pp. 570–580.
[32] R. D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Practice & Experience, 3 (1991), pp. 457–481.
[33] G. C. Fox, *A review of automatic load balancing and decomposition methods for the hypercube*, Springer-Verlag, New York, 1988, pp. 63–76.
[34] C. Farhat and M. Lesoinne, *Automatic partitioning of unstructured meshes for the parallel solutions of problems in computational mechanisms*, International Journal for Numerical Methods in Engineering, 36 (1993), pp. 745–764.
[35] H. D. Simon, *Partitioning of unstructured problems for parallel processing*, Computing Systems in Engineering, 2 (1991), pp. 135–148.
[36] A. Pothen, H. D. Simon, and K. P. Liu, *Partitioning sparse matrices with engenvectors of graphs*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 430 – 452.
[37] E. R. Barnes, *An algorithm for partitioning the nodes of a graph*, SIAM Journal for Algorithms and Discrete Methods, 3 (1982), pp. 541–550.
[38] R. B. Boppana, *An average case analysis*, in the 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, 1987, pp. 67–75.
[39] B. Hendrickson and R. Leland, *An improved spectral graph partitioning algorithm for mapping parallel computations*, SIAM Journal on Scientific Computing, 16 (1995), pp. 452–469.
[40] M. Banan and K. D. Hielmstad, *Self-organization of architecture by simulated hierarchical adaptive random partitioning*, in International Joint Conference of Neural Network (IJCNN), vol. 3, Baltimore, MD, 1992, pp. 823–828.
[41] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, *Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning*, Operations Research, 37 (1989), pp. 865 – 892.
[42] D. E. V. den Bout and T. K. M. III, *Graph partitioning using annealed neural networks*, IEEE transaction on Neural Networks, 1 (1994), pp. 192–203.
[43] E. Rolland, H. Pirkul, and F. Glover, *Tabu search for graph partitioning*, Annals of Operations Research, 63 (1996), pp. 209–232.
[44] T. N. Bui and B. R. Moon, *Genetic algorithm and graph partitioning*, IEEE Transactions on

Computers, 45 (1996), pp. 841–855.
[45] M. R. Garey, D. S. Johnson, and L. Stockmeyer, *Some simplified np-complete problems*, Theoretical Computer Science, 1 (1976), pp. 237–267.
[46] M. R. Garey and D. S. Johnson, *Computer and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.
[47] T. N. Bui and C. Jones, *Finding good approcimate vertex and edge partitions is np-hard*, Information Processing Letters, 42 (1992), pp. 153–159.
[48] V. Cardellini, M. Colajanni, and P. S. Yu, *Request redirection algorithms for distributed web systems*, IEEE Transactions on Parallel and Distributed Systems, 14 (2003), pp. 355–368.
[49] S. Floyd and V. Paxson, *Difficulties in simulating the internet*, IEEE/ACM Transactions on Networking, 9 (2001), pp. 392–403.
[50] M. Arlitt and T. Jin, *A workload characterization study of the 1998 world cup web site*, IEEE Network, 14 (2000), pp. 30–37.
[51] P. Barford and M. Crovella, *A performance evaluation of hyper text transfer protocols*, Proceeding of ACM Sigmetrics, 27 (1999), pp. 188–197.