Scheduling Multisource Divisible Loads on Arbitrary Networks

Jingxi Jia, Bharadwaj Veeravalli, *Senior Member*, *IEEE*, and Jon Weissman, *Senior Member*, *IEEE*

Abstract—Scheduling multisource divisible loads is a challenging task as different sources should cooperate and share their computing power with others to balance their loads and minimize total computational time. In this study, we attempt to address a generalized divisible load scheduling problem for handling loads from multiple sources on arbitrary networks. This problem is all the more challenging as 1) the topology is arbitrary, 2) in such networks, it is difficult to decide from which source and which route a processing node should receive loads, and 3) processing nodes must be allocated to different sources when they become available. We study two distinct cases of interest, *static* case and *dynamic* case, and propose two novel strategies, referred to as *Static Scheduling Strategy* (SSS) and *Dynamic Scheduling Strategy* (DSS), respectively. Both strategies work in an iterative fashion. In each iteration, they will use a novel *Graph Partitioning* (GP) scheme to partition the network such that each source in the network gains a portion of network resources and then these sources cooperate to process their loads. We analyze the performance of DSS using queuing theory and derive upper bounds on a load's average waiting time and a source's average queue length. We use simulation to verify the usefulness and effectiveness of SSS and DSS. Our findings reveal an interesting "load insensitive" property of SSS and also verify the theoretical upper bound of average queue length at each source in the dynamic case.

Index Terms—Divisible loads, multisource, communication delay, processing time, arbitrary network.

1 Introduction

Data-driven computation is an active area of current research fueled by the immense amount of data. Handling such data/loads on a single workstation can be quite time consuming, and hence people seek solutions in a network-based environment. One category of these loads, where there is no precedence relationship among themselves, is referred to as divisible loads. The divisible load paradigm, originated from Cheng and Robertazzi [1], has been proposed as an effective technique to schedule and process such computationally intensive loads on networks. A formal mathematical framework of this technique was provided by Bharadwaj et al. [2] and the theory was formally referred to as *Divisible Load Theory* (DLT). DLT explicitly captures the processors computation capacities and link communication delays in the problem formation, and seeks optimal or near-optimal solutions in scheduling and processing the tasks. Since its inception, the DLT paradigm has been applied to many applications including large-scale matrix-vector product [3], large-scale database search problems [4], the use of DLT paradigm with clusters of workstations [5], [6], etc. DLT researchers are also incorporating many realistic system constraints into the problem formulation, such as buffer constraints [7], [8], communication start time costs [9], release times of the processors [10], and others. Scheduling divisible loads with multiround algorithms is studied in [11], and the work [12] considers scheduling divisible loads in a resource unaware environment.

The works cited above have one common assumption, that is, they assume the initial load(s) reside on a single workstation. This means there is only one single load origin in the network. However, in many real-life applications, loads/data may be generated and produced at many different network locations. In these cases, the data require significant computation which may not be fully available at the location of the distributed data sources. This applies to many application domains. Examples include data mining of distributed scientific and medical data, processing and analysis of sensor-collected data (e.g., detecting patterns in camera images), and high-energy physics (e.g., analyzing LHC data). In these examples, there are multiple load origins/sources in the computing networks, and computation must be harnessed in the network to meet the computational demand of the applications.

However, designing an efficient multisource scheduling strategy is more difficult since multiple sources must cooperate with each other to share the resources. Because of this complexity, the multisource scheduling problem has received much less attention in the literature. Recent works [13], [14] have addressed the multisource scheduling problem on a tree network via linear programming and closed form solutions, respectively. Another work [15] studied the two sources scheduling problem on linear networks, and the work [16] consolidates the previous results. However, those works are mainly focused on networks with regular topologies, and in most cases only two load origins/sources are considered. The generalized case, scheduling multisource divisible loads on an arbitrary network, has not been rigorously studied in the DLT literature.

- *J. Jia and B. Veeravalli are with the Computer Networks and Distributed Systems Laboratory (CNDS), Department of Electrical and Computer Engineering, National University of Singapore, Singapore 117576, Singapore. E-mail: {g0500093, elebv}@nus.edu.sg.*
- *J. Weissman is with the University of Minnesota, 4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455-0159. E-mail: jon@cs.umn.edu.*

Manuscript received 20 Mar. 2008; revised 26 Nov. 2008; accepted 17 Mar. 2009; published online 3 Apr. 2009.
Recommended for acceptance by R. Eigenmann.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-03-0111.
Digital Object Identifier no. 10.1109/TPDS.2009.62.

1.1 Our Contributions and Scope of the Work

This paper is first of its kind to address the multisource divisible loads scheduling problem on an arbitrary network in the DLT domain. A similar but different problem of scheduling multiflows on arbitrary networks has been addressed using the multicommodity flow model [17], [18]. However, multicommodity flow modeling and divisible load scheduling paradigm have different concerns. In multicommodity flow problems, commodities flow from a set of known sources to a set of known sinks via an underlying network and a major concern is to seek a maximal flow. So, determining routes that provide maximal flow between sources and sinks is a key concern. However, in the DLT domain, every node is a potential sink and the connotation of "sink" as a special kind of node is not found in the DLT problem formulation. Thus, a load fraction is allowed to be processed anywhere in the system. Also note that DLT provides a discrete, fine grained control of the system, such as timing control (i.e., when a processor should send a load fraction to another processor, based on delays), while this is not the main concern with the multicommodity flow problem.

The scope of this paper is to design and analyze multisource divisible load scheduling strategies on arbitrary networks within the DLT domain. We consider two different cases of interest, the static case and the dynamic case. In the static case, we assume that no new loads will arrive to the system, while in the dynamic case, new loads may arrive as the time progresses. Since a processing node may be connected to multiple sources and may have multiple routes to a source, a fundamental issue is from which source(s) and which route(s) a processing node should receive its loads. This poses considerable challenges on designing effective strategies.

To address this issue, we propose a novel *Graph Partitioning* (GP) scheme. GP partitions the network into several totally disjoint regions, which solves the "from which source" issue, and also generates a shortest path spanning tree (SPST) for each region, which solves the "from which route" issue. We propose two novel strategies, which are referred to as *Static Scheduling Strategy* (SSS) and *Dynamic Scheduling Strategy* (DSS), one for each case. Both strategies use GP to partition the network, and balance the loads in an iterative fashion. We study the performance of our strategies both analytically and through simulation. In our study, SSS has exhibited an interesting "load insensitive" property, which will be discussed in Section 6. For DSS, we apply queuing theory to analyze the dynamical nature of DSS. We derive the upper bound of average load waiting time and average queue length, and verify the upper bound through simulation.

The paper is organized as follows: In Section 2, we introduce the network model and the problem formulation. In Section 3, we present how GP works, and describe SSS. We present the DSS in Section 4. In Section 5, we analyze DSS using queuing theory. Simulation results and discussions are in Section 6. Finally, we conclude the paper in Section 7.

2 Network Model, Assumptions and Problem Formulation

In this section, we describe the network model and present the problem we address. We consider an arbitrary connected network comprising a total of m source nodes, to which users submit loads for processing. These source nodes (or simply "sources") will share the loads either with the entire network or a portion of the network. We denote them as $S_0, S_1, \ldots, S_{m-1}$. Besides the sources, we assume that there are n processing nodes in the network. These processing nodes can receive loads from any source, and we denote them as $P_m, P_{m+1}, \ldots, P_{m+n-1}$.

We make the following assumptions in our formulation. First, both sources and processing nodes are allowed to participate in the computation process, and processing nodes can perform routing functions. Each source or processing node is equipped with a front-end processor which off-loads communication tasks of that processor. This enables the computation and communication to be carried out concurrently [19]. However, we assume that no node is equipped with multiple separate ports for communication, and therefore simultaneously transmitting or receiving is not allowed. We also assume that sources can share information with each other through messages, and we neglect any overheads incurred by transmitting such short messages. Further, a linear cost model for communication and computation is adopted as in the literature. Therefore, communication delay and computation time is assumed to be linearly related to the load size.

Problem Statement: *Given an arbitrary graph* $G = <V, E>$, *with* $V = m + n$, *where* m *equals the number of sources and* n *equals the number of processing nodes, how do we schedule and process loads submitted by the source nodes in the system such that the total processing time is minimized.*

We consider two distinct cases of interest—the static case and the dynamic case. For the static case, we assume that in the network there are m divisible loads $L_0, L_1, \ldots, L_{m-1}$ residing on m sources, respectively. We assume that no additional loads will arrive. For the dynamic case, we assume that each source has an independent load inflow. Therefore, new loads may be expected to arrive at any point in time and the network should accommodate the new arrivals dynamically.

2.1 Problem Formulation

Two different approaches [16] are possible to tackle this problem. One is based on "superposition," wherein all or part of the processing nodes will receive multiple fractions of loads from different sources and the total load that each processing node received will be balanced according to its computation capacity. The other approach is referred to as "network partitioning," wherein the entire network will be partitioned into several nonoverlapping regions centered at each source, respectively, and each source will only dispatch its load to its own region. Both techniques have advantages and disadvantages. Under "network partitioning," since the entire network is partitioned into nonoverlapping regions, each source can carry out load dispatching separately without interfering with each other. However, the challenge lies in partitioning the network into regions where each region's equivalent computation power (defined in [2]) is exactly proportional to this region's load size. In most cases, we cannot strike a perfect balance across the network. On the other hand, under the "superposition" technique, each processing node can receive loads from

several sources, and hence it is easier to balance the load across the network. However, because of the communication contention problems, exercising control of "superposition" is much more complicated than "network partitioning," and may induce large overheads.

In the divisible load context, all load fractions are homogenous, and a node that receives multiple fractions of load from different sources can receive a single fraction of load, which equals the summation of the multiple fractions, from the "nearest" source. Therefore, we adopt "network partitioning" technique[1] to schedule and process the loads.

However, as mentioned above, designing an efficient strategy to partition the graph into several regions such that each region's optimal *equivalent computation power* is proportional to the region's load size is very difficult. The reason is that one must first solve a (equivalent) subproblem: *Given a region, how do we identify this region's optimal equivalent computation power?* It has been shown in the DLT literature that the optimal solution of scheduling divisible loads on an arbitrary graph occurs on a spanning tree of the graph. Thus, to identify a region's optimal equivalent computation power, we must find the best/optimal spanning tree first. Unfortunately, finding the best/optimal spanning tree for divisible load distribution on a graph is proven to be NP-hard in [20].

Therefore, in this paper, we will demonstrate that our proposed GP works efficiently in the sense that it partitions the network into several regions and generates an *SPST* for each region simultaneously. Our two algorithms, SSS for the static case and DSS for the dynamic case, will use GP to partition the network and then schedule the loads.

3 Static Scheduling Strategy

Before we introduce SSS, we will first illustrate how GP works. For the ease of presentation, we define an *ordered communication delay two-tuple* (C_{ki}, i) which captures the cumulative communication delay from processing node P_k to source S_i. As there are m sources in the network, each processing node has m two-tuples. We can define their relations as follows: When $C_{ki} > C_{kj}, (C_{ki}, i) > (C_{kj}, j)$. When $C_{ki} = C_{kj}$, then $(C_{ki}, i) > (C_{kj}, j)$, iff $i > j$. Since each source has a unique subscript, according to our definition, each processing node can locate a unique source which has smallest two-tuple (i.e., smallest cumulative communication delay). We denote this source as target source to the corresponding processing node. Further, we also define π_{ij} as the shortest path (in term of communication delays) from S_i (or P_i) to S_j (or P_j).

Graph partitioning scheme. In our approach, the given graph is divided into m regions $Region_0, Region_1, \ldots Region_{m-1}$, centered at $S_0, S_1, \ldots, S_{m-1}$, respectively. An arbitrary processing node P_i is attached to its target source by the shortest path (path with smallest communication delay). We observe that GP intuitively uses each processing node effectively. This is because what determines the real computation capacity of a node in a network is not only the computation speed of this processor, but also its communication delay to the source. In GP, all the processing nodes are attached to their target sources by the shortest path and, hence, from processing nodes' perspective they have been used efficiently. We define the following.

Totally disjoint regions: Totally disjoint regions mean that any two regions have 1) no common node, 2) no common link, and 3) no intersection. Notice that 1) and 2) do not imply 3). Even without common nodes and links, two regions may still intersect with each other. For example, suppose P_x belongs to $Region_i$, and P_y belongs to $Region_j$, respectively. However, P_y may be connected to S_j through P_x (i.e., P_x provides routing service for $Region_j$), so $Region_i$ and $Region_j$ still intersect with each other. Thus, for two regions to be totally separable/disjoint, they must satisfy 1), 2), and 3) simultaneously. Now we state the following.

Theorem 1. *Using GP the graph is divided into m totally disjoint regions.*

Proof. In order to realize the proof we proceed as follows: We realize that 1) is immediately apparent, since every processing node has a unique smallest ordered communication delay two-tuple. However, to complete the proof including 2) and 3), we need to prove the following lemma first. □

Lemma 1. *Suppose P_i is attached to S_j, and π_{ij} is the shortest path (with respect to communication delay) from P_i to S_j. Then, all the processing nodes belonging to π_{ij} are also attached to S_j.*

Proof. The proof is by contradiction. Suppose one processing node P_k belongs to π_{ij} and is not attached to S_j, but is attached to another source S_x. Thus, according to GP, for P_k, we have $(C_{kj}, j) > (C_{kx}, x)$. Notice that the path from P_x to P_i has a constant communication delay, denoted as $C_{\pi_{xi}}$. Then, for P_i, we have $(C_{kj} + C_{\pi_{xi}}, i) > (C_{kx} + C_{\pi_{xi}}, x)$, i.e., $(C_{ij}, j) > (C_{ix}, x)$. This means that P_i should also be attached to S_x, which contradicts our assumption. Therefore, Lemma 1 is proved.

Next, we use Lemma 1 to complete the proof of separability of the regions including 2) and 3) of Theorem 1. According to GP, each processing node is attached to its target source by the shortest path. Suppose two paths π_1 and π_2, which belong to two different regions $Region_i$ and $Region_j$, respectively, intersect with each other at P_k. Then, from Lemma 1 we know P_k belongs to $Region_i$ and also $Region_j$, which is impossible. Therefore, any two paths belonging to two different regions have no intersection, and hence any two regions have no intersection or common links. Hence, the proof. □

From the above proof, we know that by using GP, the graph can be divided into m disjoint regions. Further, since in GP all processing nodes in the same region are attached to the corresponding source by the shortest path, we automatically generate one SPST for each region. This is a very useful characteristic of GP, since the optimal solution of scheduling divisible load for an arbitrary connected graph occurs on a spanning tree of the graph. In our context, after applying GP, each source can directly dispatch load to its SPST, using a similar RAOLD-OS

1. Since in our problem context the network has an arbitrary graph topology, "network partitioning" is actually "graph partitioning."

strategy [21]. Therefore, GP actually performs two tasks together. It effectively divides the graph into m disjoint regions and at the same time it generates one SPST for each region. We use the shortest path for constructing a spanning tree because each processing node's communication delay is guaranteed to be a minimum. Thus, compared to other well-known spanning trees, such as shortest hop spanning tree, minimum spanning tree [21], and robust spanning tree [22], SPST usually admits better performance.

Now we will introduce how SSS works. SSS progresses in an iterative fashion. At the beginning of the first iteration, SSS will apply GP to partition the graph. After the network is partitioned into m totally disjoint regions, each source will compute the "equivalent computation power" for its own region based on the SPST and this process of obtaining an equivalent computation power is described in [2]. Notice that since we do not take into account each source's load size when we partition the network, each region's "equivalent computation power" is not proportional to this region's load size. We may expect that regions will complete processing their respective loads at different time instants. Therefore, in the first iteration, to balance the computation power across all the regions, the amount of load that will be consumed for processing by a region will be altered proportionally. Suppose when each source dispatches and processes its load only on its own region, the finish time of $L_0, L_1, \ldots, L_{m-1}$ are $T_0, T_1, \ldots, T_{m-1}$, respectively, and suppose $i = argmin\{T_j\}$, i.e., $Region_i$ has the smallest finish time. To achieve balance, any region other than $Region_i$, say $Region_j$, will only consume L_jT_i/T_j amount of load in the first iteration, using a similar RAOLD-OS strategy.

In the first iteration, the whole L_i is consumed by $Region_i$, and hence no load remains in S_i. However, other sources will have some amount of load remaining, and the amount of load L'_j remaining with source node S_j is

$$L'_j = L_j * \frac{T_j - T_i}{T_j}. \quad (1)$$

Therefore, the second iteration will start with $m-1$ remaining loads $L'_0, \ldots L'_{i-1}, L'_{i+1}, \ldots, L'_{m-1}$ residing on $m-1$ sources $S_0, \ldots S_{i-1}, S_{i+1}, \ldots, S_{m-1}$, respectively. Then, SSS will apply GP again to partition the graph into $(m-1)$ regions. Notice that this process actually is a reallocation of processing nodes, which originally belong to $Region_i$, to other regions. Those processing nodes which belong to a region other than $Region_i$ remain in that region. As in the first iteration, the region with the smallest finish time will consume the whole remaining load, while other regions will only consume a proportional amount of load, and hence the third iteration will start with $(m-2)$ sources and remaining loads. Obviously, SSS will come to a halt after m iterations. Further, as long as a region is busy, its equivalent computation power will not decrease. Thus, in SSS, the processing of load L_i will complete within T_i. The total processing time T_{sss} of the entire network, defined as time difference between the start time and the time instant when the last remaining load has been processed, is

$$T_{sss} \leq \max\{T_i, i = 0, 1, \ldots, m-1\}. \quad (2)$$

We observe two issues here. First, in SSS, within each iteration, "network partitioning" technique is used to dispatch and process the loads. However, when we look at the entire process, a processing node may receive loads from different sources, and hence SSS also has an "superposition" characteristic. Therefore, SSS can be viewed as having a "hybrid" property.

Second, when implementing GP, it can either be the sources that can initiate the processing or the processing nodes.[2] In a source initiating scheme, each source will construct a shortest path spanning tree simultaneously, using Dijkstra's Algorithm or the Bellman-Ford Algorithm [23]. Then, all the sources share information with each other, and hence each source can identify the processing nodes which have the smallest communication delay to itself. On the other hand, if processing nodes initiate the algorithm, each processing node will simultaneously compute its shortest path weight (communication delay) to each source using Dijkstra's Algorithm or Bellman-Ford Algorithm, and then choose the target source and report the shortest route to the target source. To reduce redundant computation, initially, each processing node can maintain a list of shortest paths and their weights to each source. Then, as long as a source completes its load, its processing nodes (nodes within its region) can quickly identify the next source it should be attached to, and hence reduce overheads.

4 Dynamic Scheduling Strategy

Now we tackle a more realistic situation wherein each node is more independent in its operation and each source has an independent and dynamic load inflow. To accommodate the newly arrived loads, we attempt to extend SSS. In this DSS, at the beginning of each iteration, each source will check which sources in the network currently have loads to be processed. This can be done by exchanging messages among the sources. Then, sources having loads to process, will apply GP to partition the network. In a manner similar to SSS, the region with the smallest finishing time will consume the entire load, while other regions will only consume a proportional amount of load. After the current iteration, the sources will repeat the above process for every load that arrives to the system.

There are two major concerns here. First, a newly arrived load at S_i will be stored in the buffer until all previous loads in S_i have been processed. Second, unlike in SSS, where each active region's "equivalent computation power" will only increase, in DSS, it may also decrease. This is because a new load may arrive at a previously idle source, and this source will reclaim the resource which initially belongs to its region at the beginning of the next iteration. Therefore, the "equivalent computation power" of a currently active region fluctuates as the time progresses.

Although in DSS each region's "equivalent computation power" fluctuates, we can still attempt to derive the *upper bound* of processing time of a given load.[3] When all sources in the network are busy, each region will occupy certain "domains" in the network. We refer to such domains as the

2. In the literature, these are commonly referred to as sender initiated and receiver initiated approaches.

3. Time difference between the instant at which the load is scheduled by the source and the time instant at which the load has been completed.

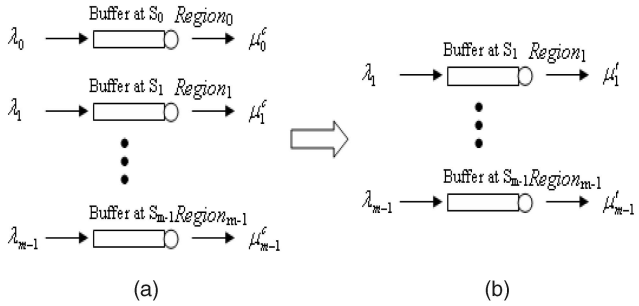


Fig. 1. Network models. (a) Network model when all regions are busy. (b) Network model when only $Region_0$ is idle.

"critical domains" to the corresponding regions. Notice that a region's "critical domain" is composed of all the processing nodes which have smaller communication delay to this region's source than any other sources. For a certain source, its "critical domain" will always be attached to it, as long as this source is busy. We denote the "equivalent computation power" of $Region_i$'s "critical domain" as E_i^c. Further, since we adopt a linear cost model, the processing time of a load is linearly related to the load size. Therefore, suppose the load L_i at S_i is processed with k installments $L_i^0, L_i^1, \ldots, L_i^{k-1}$, and the average computation power is $\bar{E}_i$, we have

$$\begin{aligned} T(\bar{E}_i, L_i) &= T\big(\bar{E}_i, L_i^0 + L_i^1 + \cdots + L_i^{k-1}\big) \\ &= T\big(\bar{E}_i, L_i^0\big) + T\big(\bar{E}_i, L_i^1\big) + \cdots + T\big(\bar{E}_i, L_i^{k-1}\big) \\ &\le T\big(E_i^c, L_i^0\big) + T\big(E_i^c, L_i^1\big) + \cdots + T\big(E_i^c, L_i^{k-1}\big) \\ &= T\big(E_i^c, L_i\big) = E_i^c L_i, \end{aligned} \tag{3}$$

where $T(E, L_i)$ denotes the processing time of load L_i with computation power E.

Notice that (3) gives an upper bound of one load's processing time. However, a load may not be able to be processed immediately when it arrives, and hence the actual time the load spends in the network may be longer. Further, since a newly arrived load will be stored in the buffer until its previous loads have been processed, each source should have adequate buffer space to hold new arrival loads. Therefore, it will be more appropriate if we perform queuing analysis for understanding the performance of DSS in the next section.

5 Analysis of DSS

Suppose each source has independent Poisson arrival loads, and the arrival rate at S_i is λ_i. Further, we assume that the load size is exponentially distributed with parameter $\mu_i^c E_i^c$. Notice that when a region has fixed computation power, the processing time of the loads is also exponentially distributed. Therefore, when all regions are busy (and thus each region is processing its load within its critical domain only), we can map each region to a M/M/1 queue [24], and the service rate for $Region_i$ is μ_i^c, as shown in Fig. 1a.

However, in a dynamic situation, the service rate for each region is not constant. At any time instant, several regions may become idle and their computation power will be reallocated to other regions. For instance, when $Region_0$ is idle while other regions are busy, the network model is shown in Fig. 1b. Notice that the arrival rate λ is the same for the above two cases, but $\mu_i\prime = \frac{E_i'}{E_i^c}\mu_i^c$, where E_i^c denotes the equivalent computation power of $Region_i$'s critical domain and E_i' denotes $Region_i$'s equivalent computation powers when $Region_0$ is idle. Actually, there are 2^m different cases, where m is the number of sources. In all cases, for a given region, arrival rates remain the same, but service rates are different. Notice that among all cases, when S_i occupies the entire network (i.e., when all other regions are idle), $Region_i$ will have the maximum service rate. We denote the maximum service rate for $Region_i$ as μ_i^{max}.

Now, we consider the entire network as a system with m inflows of loads and the system will process these loads at a certain rate. However, if the aggregated load inflow rate exceeds the service rate of the entire system, then system becomes unstable, and the number of loads left in the system will increase to infinity as the time progresses. Therefore, the key question to address is as follows: "*Given a network, if we know the load arrival rate as well as the load size distribution at each source (i.e., λ_i and each region's service rates in different cases are known), how can we decide whether this network can manage these loads or not, and what is the average queue length at each source and the average waiting time of a load?*" To address these questions, we will analyze our network for three important cases.

Case 1. $\forall i, \lambda_i < \mu_i^c$. This is considered as a stable case wherein arrival rates are less than the processing rates, which implies that each source can easily manage its own loads using its critical domain only.

According to DSS, an idle source node S_i will release its computation capacity. Any load arriving during S_i's idle time will wait for a certain amount of time until S_i regains its computation capacity, and then S_i will start to dispatch and process the load. Therefore, we can map $Region_i$ to a M/G/1 queue with vacations[4] [24]. Let ζ_i denote the distribution of service time at $Region_i$, and ν_i denotes the distribution of $Region_i$'s vacation time. Then, average waiting time of loads at S_i's buffer (denoted as T_i^w) is given by

$$T_i^w = \frac{\lambda_i E\big[\zeta_i^2\big]}{2(1-\lambda_i E[\zeta_i])} + \frac{E\big[\nu_i^2\big]}{2E[\nu_i]}. \tag{4}$$

However, in our context, each region's service rate and vacation time are coupled with other regions in the network, and hence it is more appropriate to view the entire network as m coupled M/G/1 queues with vacations. In this case, it may be noted that determining the exact value of T_i^w is very difficult. To see this, consider the simplest case where there are only two regions ($Region_0$ and $Region_1$) in the network. To compute $E[\zeta_i], E[\zeta_i^2], E[\nu_i]$, and $E[\nu_i^2]$ $(i = 0, 1)$ in (4), one needs to know the probability when $Region_0$ is busy while $Region_1$ is idle, the probability when $Region_0$ is idle while $Region_1$ is busy, the probability when both regions are busy, and the probability when both regions are idle. This requires us to solve an infinite three-dimensional Markov Chain, as shown in Fig. 2. In the figure, state (ij) denotes i loads in S_0 and j loads in S_1, while both S_0 and S_1 are busy. State $(i'j)$(or $(ij\prime)$) denotes that there are i loads in S_0 and j loads in S_1, while S_0 is idle(or busy) and S_1 is busy(or idle).

4. When all other regions are idle, the vacation time can be viewed as arbitrarily small.

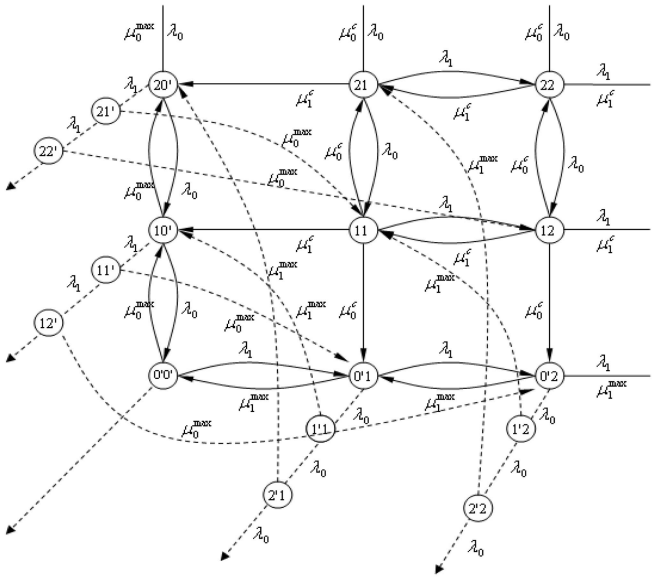


Fig. 2. Markov chain of two regions case.

The Markov Chain shown in Fig. 2 is very complicated to solve. Further, the complexity of the problem increases dramatically as the number of sources increases and makes it complex to derive an exact value of T_i^w. Thus, in this paper, we will attempt to derive the upper bound on T_i^w. In the derivation, we will use an important property of the exponential distribution—the combination property [24], which is stated as follows:

Theorem 2. *Random variables $x_1, x_2, \ldots, x_k$ are exponentially distributed with parameters $u_1, u_2, \ldots, u_k$. Let random variables $Y = \min\{x_1, x_2, \ldots, x_k\}$. Then, Y is also exponentially distributed with parameter $u = u_1 + u_2 + \cdots + u_k$.*

Using Theorem 2, we can identify the distribution of ν_i and find the worst case $E[\nu_{iworst}]$ and $E[\nu_{iworst}^2]$. Suppose when S_i becomes idle, there are k busy regions in the network. Because of the memoryless property of the exponential distribution [24], all the k regions' remaining processing time of their loads are exponentially distributed with parameters $\mu_0, \mu_1, \ldots, \mu_{k-1}$. It may be noted that the $Region_i$'s vacation time is the minimum remaining processing time among the k busy regions. According to Theorem 2, we know that the $Region_i$'s vacation time ν_i is also exponentially distributed, with parameter $\mu = \mu_0 + \mu_1 + \cdots + \mu_{k-1}$. Notice that as k becomes smaller (i.e., less regions are active), μ also becomes smaller. Therefore, when only the region with smallest μ_j^{max} is active, $Region_i$'s vacation time distribution has the smallest value $\mu = \mu_j^{max} = \min\{\mu_0^{max}, \mu_1^{max}, \ldots, \mu_{i-1}^{max}, \mu_{i+1}^{max} \ldots, \mu_{m-1}^{max}\}$. In this case, $Region_i$ has the largest average vacation time, and hence loads at S_i have largest waiting time in the buffer. We have

$$E[\nu_{iworst}] = \frac{1}{\mu_j^{max}}, \tag{5}$$

$$E[\nu_{iworst}^2] = \frac{2}{(\mu_j^{max})^2}. \tag{6}$$

Further, we notice that as long as $Region_i$ is busy, its service rate is larger than or equal to μ_i^c. Since the loads waiting time is inversely related to the $Region_i$'s service rate, the upper bound of T_i^w is

$$T_i^w = \frac{\lambda_i E[\zeta_i^2]}{2(1-\lambda_i E[\zeta_i])} + \frac{E[\nu_i^2]}{2E[\nu_i]} \le \frac{\lambda_i/\mu_i^c}{\mu_i^c - \lambda_i} + \frac{1}{\mu_j^{max}}. \tag{7}$$

Then, applying *Little's Theorem* [25], we can derive the average number of loads in the S_i's buffer (denoted as Num_i^{ave}), which is

$$Num_i^{ave} = \lambda_i T_i^w \le \lambda_i \left(\frac{\lambda_i/\mu_i^c}{\mu_i^c - \lambda_i} + \frac{1}{\mu_j^{max}} \right). \tag{8}$$

Since load size at S_i is exponentially distributed with a parameter $\mu_i^c E_i^c$, the average load size is given by $\frac{1}{\mu_i^c E_i^c}$. Thus, the average queue length at S_i (denoted as Q_i^{ave}) is bounded by

$$Q_i^{ave} = Num_i^{ave}/E_i^c\mu_i^c \le \lambda_i \left(\frac{\lambda_i/\mu_i^c}{\mu_i^c - \lambda_i} + \frac{1}{\mu_j^{max}} \right) \Big/ E_i^c\mu_i^c. \tag{9}$$

Equation (9) gives us considerable hints on how much buffer should be assigned to each source when designing the system. To reduce the probability of dropping loads when the buffer is full, one should assign a larger buffer size than Q_i^{ave} derived from (9) (for example, two times larger than Q_i^{ave}), to S_i. However, since we have adopted some approximations to derive (9), in some cases Q_i^{ave} may not give a tight estimation on real actual queue length. This behavior is carefully studied and discussed in the next section.

Case 2. $\forall i, \lambda_i \ge \mu_i^c$ **or** $\exists\, i, \lambda_i > \mu_i^{max}$. In this case, the network cannot manage these loads and is *critically stable*. The average queue length of the network and average waiting time of loads are expected to grow to infinity, as time progresses. Therefore, in this situation, one must reduce load arrival rates or discard low priority loads at one or more of the sources.

Case 3. $\forall i, \lambda_i < \mu_i^{max}$ **and** $\exists i, \lambda_i \ge \mu_i^c$ **and** $\exists j, \lambda_j < \mu_j^c$. This case is more difficult than the above two cases. In this case, some regions cannot handle their loads by using their critical domains only, but by "borrowing" computation power from other regions, these regions may be able to handle their loads. Thus, the problem is *"whether the regions with excess resources can render enough computation power to other regions."* Obviously, addressing this problem is extremely complex. For this case, we attempt to study the simplest two regions case, which reveals some basic issues of the posed problem.

Consider that there are two regions in the network, with $\mu_0^c \le \lambda_0 < \mu_0^{max}$ and $\lambda_1 < \mu_1^c < \mu_1^{max}$. Now, the key question is that whether $Region_0$ can borrow enough computation capacities from $Region_1$ to accommodate its excess loads. Consider the *boundary* situation, i.e., $Region_0$ can borrow "just enough" resources from $Region_1$. In this case, $\lambda_0 = \bar{\mu_0}$, where $\bar{\mu_0}$ denotes $Region_0$'s average service rate. From $Region_1$'s perspective, it can be mapped to a M/M/1 queue with vacations, and vacation time is exponentially distributed with parameter μ_0^{max}. Notice that vacation time will not affect

the *idle ratio*[5] of $Region_1$ (denoted as R_1^{idle}), and hence R_1^{idle} is equal to the idle ratio of a M/M/1 queue with the same service rate and arrival rate, but without vacations, which is

$$R_1^{idle} = \left(\mu_1^c - \lambda_1\right)/\mu_1^c. \tag{10}$$

From $Region_0$'s viewpoint, it has the maximum service rate μ_0^{max} in R_1^{idle} ratio of time, and has a service rate of μ_0^c in the remaining $1 - R_1^{idle}$ ratio of time. Hence, the expectation of its service rate is

$$\bar{\mu_0} = \frac{\mu_0^{max}(\mu_1^c - \lambda_1)}{\mu_1^c} + \frac{\mu_0^c \lambda_1}{\mu_1^c}. \tag{11}$$

Therefore, if $\lambda_0 < \bar{\mu_0}$, this network is able to manage these loads. Otherwise, this network cannot handle this amount of loads.

As in Case 1, one can apply (9) to estimate the upper bound of Q_1^{ave} (the average queue length at S_1), and as λ_0 approaches $\bar{\mu_0}$ (given by (11)), Q_1^{ave} approaches the upper bound. When $\lambda_0 \geq \bar{\mu_0}, Q_1^{ave}$ is exactly equal to $\lambda_1(\frac{\lambda_1/\mu_1^c}{\mu_1^c - \lambda_1} + \frac{1}{\mu_0^{max}})/E_1^c\mu_1^c$ (as per (9)). For $Region_0$, when $\lambda_0 \geq \bar{\mu_0}, Q_0^{ave}$ goes to infinity. When $\mu_0^c < \lambda_0 < \bar{\mu_0}$, though we cannot directly use (9) to calculate the upper bound of Q_0^{ave}, we can use $\bar{\mu_0}$ instead of μ_0^c in (9) to estimate the approximate value of Q_0^{ave}.

Similarly, when there are more than two regions in the network, we have to compute how much computation power the regions with excess resource can borrow from other regions which cannot handle their loads alone. Unfortunately, solving this problem requires solving the similar Markov Chain[6] as shown in Fig. 2. This remains an open problem.

6 Performance Evaluation

To characterize the network used for simulation in this section, we define several network parameters that are widely used in the DLT literature [2].

1. z_i: Communication speed parameter. It is the ratio of the time taken to transmit a unit amount of data through the link l_i to the time taken by a standard link.
2. w_i: Computation speed parameter. It is the ratio of the time taken to compute a unit amount of data by the S_i (when $i \leq m-1$) or P_i (when $i \geq m$) to the time taken by a standard processing node. A standard processing node or link can be any processing node or link that is referenced in the system.
3. T_{cm}: Communication intensity constant. It equals the time taken by a standard link to transmit a unit of the load.
4. T_{cp}: Computation intensity constant. It equals the time taken by a standard processing node to compute a unit of the load.

6.1 Performance of SSS

In this section, we will study the performance of the SSS. We compare the performance of SSS with a strategy referred to as *Sequential Dispatching Strategy* (SDS).[7] SDS works as follows: Consider a network with m loads $L_0, L_1, \ldots, L_{m-1}$ residing on m sources $S_0, S_1, \ldots, S_{m-1}$, respectively. In SDS, S_0 will first dispatch L_0 to the entire network based on an SPST of the network using a similar RAOLD-OS strategy, while other sources temporarily hold their loads. Then, after L_0 has been processed, S_1 will dispatch L_1 to the entire network. The above process continues until all loads have been processed.

As we can see from the above description, SDS is simple in nature. Further, we notice that if communication delay can be neglected (when all links in the network are sufficiently fast), SDS and SSS will have exactly the same performance. Suppose there are m loads in the network, the total processing time T_{total} for both SDS and SSS would be

$$T_{total} = (L_0 + L_1 + \cdots + L_{m-1}) \cdot E(w) T_{cp}, \tag{12}$$

where $E(w)$ is the equivalent computation capacity of the entire network. However, in the presence of communication delay, SSS and SDS will show different performances. We conduct experiments to study how SSS and SDS will react to communication delay. Our experiments reveal certain interesting characteristics of SSS.

In our experiments, the network has an arbitrary graph topology generated randomly with a specified number of nodes and link connectivity probabilities.[8] The computation speed parameters of sources and processing nodes w is uniformly distributed among $[1, 10]$, and both T_{cm} and T_{cp} are set to be 1. We simulate different network scenarios (tightly coupled and loosely coupled) by assigning different distributions of the communication speed parameters. We assume that each source in the network has an amount of load $L = 10{,}000{,}000$. We will show the effect of load size on our strategies later in this section. Further, we neglect the start-up cost (the time needed to run the graph partitioning scheme and calculate the load distributions), as this cost is a prescheduling and one-time cost only, and negligible compared to the real processing time.

We first study networks with 20 nodes. To simulate the characteristics of a tightly coupled network, z is set to be uniformly distributed among $[0, 0.5]$, and to simulate the characteristics of a loosely coupled network, z is set to be uniformly distributed among $[1, 2]$. We vary the number of source nodes[9] in the network from 1 to 10, and the corresponding total processing time of SSS and SDS is shown in Figs. 3a and 3b. From these figures, we observe that SSS outperforms SDS, and when the communication delay is large, SSS gains a significant speedup against SDS. This is expected since in the presence of communication delays, SSS utilizes the computation power of sources and processing nodes much more efficiently than SDS.

Further, we notice that the total processing time of SDS is approximately linearly related to the number of sources in both loosely coupled networks and tightly coupled networks, as shown in Figs. 3a and 3b. SSS also exhibits the similar linear

5. Idle ratio is defined as ratio of the time when the region is idle.

6. Similarly, the complexity of the Markov Chain increases dramatically as number of regions grows.

7. As of this date, there are no multijob strategies for scheduling divisible loads on arbitrary graphs in the literature.

8. In our experiments, the link probability of a direct link between a pair of nodes is set to 0.4.

9. Each time new source nodes are randomly generated, while previous source nodes are retained.

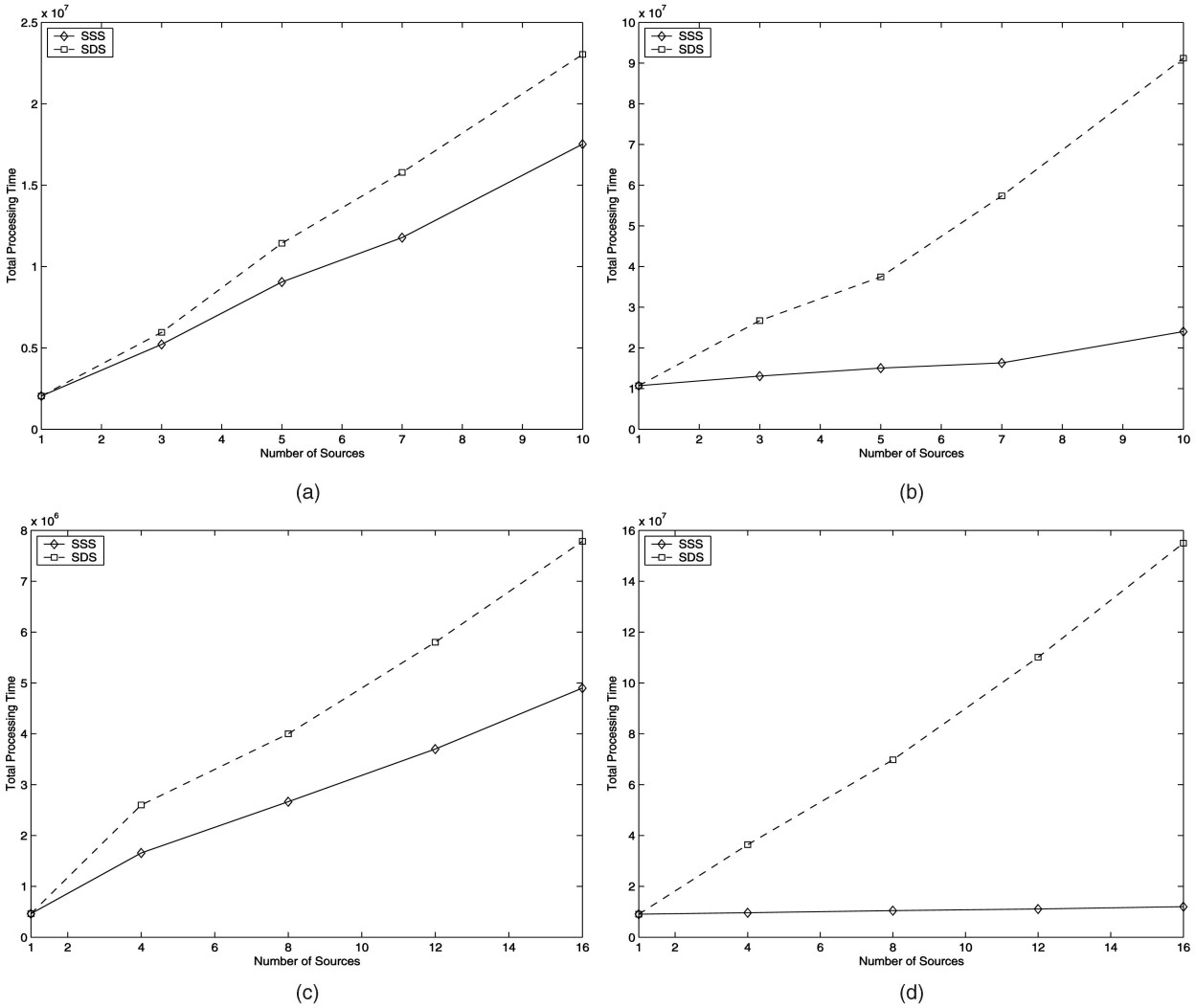


Fig. 3. Experiment results for the static case. (a) Tightly coupled network with 20 nodes. (b) Loosely coupled network with 20 nodes. (c) Tightly coupled network with 150 nodes. (d) Loosely coupled network with 150 nodes.

relationship in tightly coupled networks. However, SSS shows a very interesting characteristic in loosely coupled networks. As shown in Fig. 3b, provided the number of sources is smaller than some threshold, increasing the number of sources (i.e., increasing the number of loads in the network) does not affect the total processing time significantly. Actually, as the number of sources increases from 1 to 7 (i.e., total amount of loads increases by 600 percent), the total processing time of SDS increases by 662 percent (2.1×10^6-1.6×10^7) and 418 percent (1.1×10^7-5.7×10^7) in tightly coupled networks and loosely coupled networks, respectively. In the same case, the total processing time of SSS also increases by 471 percent (2.1×10^6- 1.2×10^7) in the tightly coupled networks, but it increases only by 45 percent (1.1×10^7-1.6×10^7) in the loosely coupled networks. We refer to the above SSS' characteristic as the "*load insensitive*" property because the total processing time of SSS seems "insensitive" to the increase of number of loads. Notice that this property can only be observed in a relatively loosely coupled network.

The above load insensitive property can be explained by a "*Nearest Nodes Dominance*" effect, which is stated as follows: *In the presence of communication delays, for a given region, the source and its "nearest" nodes dominate this region's computation capacity.* Here, "nearest" is in terms of small communication delay. This is because in our strategy the load will be scheduled according to nodes' computation speeds and communication delays to the source. In such scheduling, the "farthest" nodes (i.e., have largest communication delays) tend to receive fewer amount of load, while the nearer nodes tend to receive larger amount of load. If a farther node is deprived from a region, only a small amount of excess load needs to be shared by the rest nodes. Thus, the region's computation capacity is less affected. On the other hand, if a nearer node is deprived from the region, a large amount of excess load needs to be processed, and hence the region's computation capacity will decrease significantly.

Now, let us see why SSS exhibits the load insensitive property. The total processing time of SSS is the maximum finish time of all loads, and its upper bound is shown in the Inequality (2), where the T_is are determined by the respective critical domains' computation power. Notice that when the sources are sparse and communication delay is relatively large, each source's critical domain contains almost all its

"nearest" nodes. Therefore, because of the nearest nodes dominance effect, T_i is very close to the real processing time of L_i. Further, when adding a new source into the network, as long as sources remain sparse, it is highly probable that the new source will not deprive other sources' "nearest" nodes. Therefore, previous existing critical domains' computation capacities will not decrease significantly, and hence the total processing time is less affected.

From the above discussion, we know that there are two prerequisites for SSS to exhibit the load insensitive property. First, the communication delay of the network should be relatively large. This explains why in the tightly coupled network, the total processing time of SSS increases linearly with the number of sources, as shown in Fig. 3a. Second, the sources in the network should be sparse enough, i.e., geographically well distributed. When the number of sources exceeds some threshold, we should observe a sharp increase in the total processing time as the number of sources increases further. As shown in Fig. 3b, when the number of sources increases from 7 to 10, the total processing time increases from 1.6×10^7 to 2.4×10^7, almost triple the increment as compared to when the number of sources increases from 1 to 4 or from 4 to 7. However, notice that the threshold could be varied. For a larger network, SSS should be able to sustain the load insensitive property for a larger number of sources.

To verify this, we then conduct another set of experiments on a much larger network with 150 nodes. The network parameters are the same as previous experiments—w is uniformly distributed among $[1, 10]$, and z is uniformly distributed among $[0, 0.5]$ for tightly coupled network and among $[1, 2]$ for relatively loosely coupled network. The results are shown in Figs. 3c and 3d. We observe that for the loosely coupled case the plot of SSS is almost flat, even when the number of sources exceeds 7. Actually, as the number of sources increases from 1 to 20, the total processing time only increases from 0.91×10^7 to 1.25×10^7. However, when the number of sources increases further to 40, the total processing time increases disproportionately to 2.3×10^7.

Further, we notice that as the network size grows, the total processing time for tightly coupled network decreases significantly, but the total processing time for loosely coupled network does not change significantly. Comparing Figs. 3b and 3d, we find that for the single source case, the total processing time only decreases from 1.1×10^7 to 0.91×10^7, as the network size grows from 20 to 150 nodes. This indeed verifies the nearest nodes dominance effect.

Finally, it should be noted that since we adopt a linear cost model, the change of the initial load size does not affect the above observations. Consider the 150 nodes loosely coupled network, we vary each region load size from 5,000,000 to 15,000,000, and the total processing time of SSS with respect to different number of sources are plotted in Fig. 4. From the figure, we observe that the total processing time of SSS increases linearly with the load size.

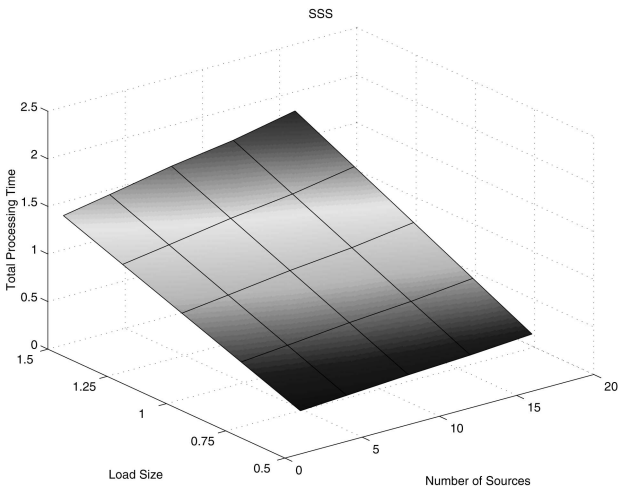


Fig. 4. Total processing time of SSS with different load size and number of sources.

6.2 Performance of DSS

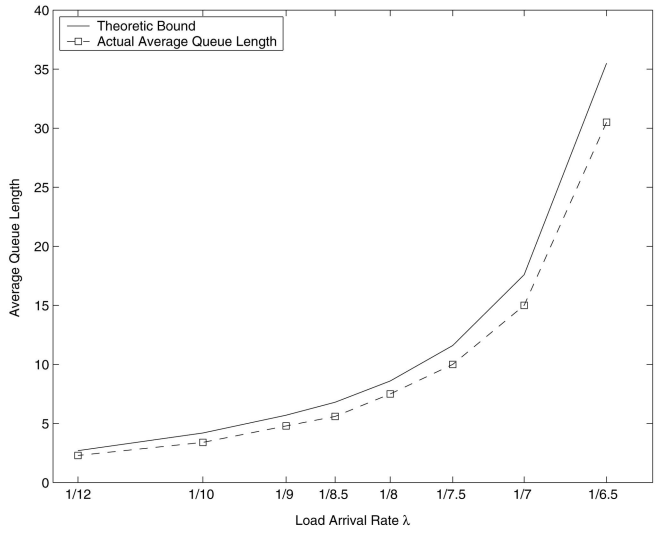
Now, we will study the performance of DSS. Since DSS is a natural extension of SSS, its usefulness and effectiveness are shown in the above section. Therefore, in this section, we mainly focus on the dynamic nature of DSS—the average queue length at each source. Notice that as long as we know the average queue length, applying Little's Theorem can easily yield other performance metrics.

We adopt the assumption made in Section 5, that is, the arrival of loads follow a Poisson distribution and load size is exponentially distributed. Further, in our simulation, for any $Region_i$, we always let $\lambda_i < \mu_i^c$, which corresponds to Case 1 in Section 5. It is because the other two cases are either trivial (for Case 2) or too complex (for Case 3). Under the above constraints, an upper bound of each region's average queue length is given in (9). Below, we conduct experiments to study the actual average queue length of each region.

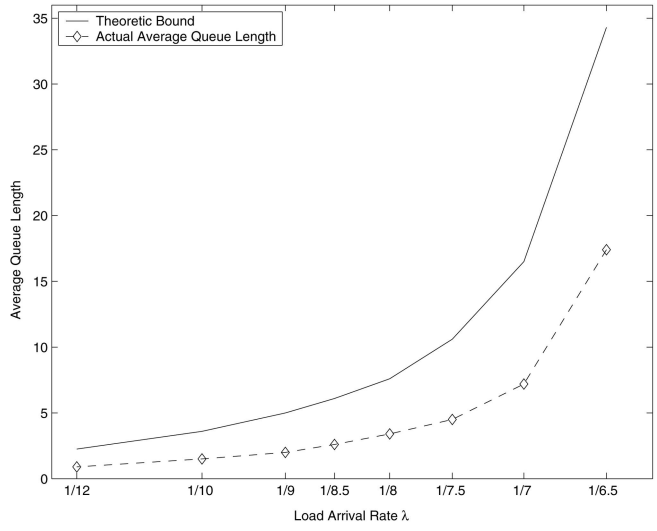
First, we consider networks with symmetric architecture and three sources. Similar to the static case, we generate two types of networks—loosely coupled and tightly coupled networks. In the loosely coupled network, because of the presence of large communication delays, a region can only get a small amount of computation power from other idle regions. On the other hand, in the tightly coupled network, since the communication delays are small, a region can get relatively larger amount of computation power from other idle regions. The respective equivalent computation power for each region in different cases is shown in Table 1.

TABLE 1
Regions' Equivalent Computation Capacities for Symmetric Networks

Network Type	$E_i^c, i = 0, 1, 2$	$E_i', i = 0, 1, 2$, when only $Region_i$ is idle.	E_i^{max}
Loosely-coupled Network	2	1.8	1.65
Tightly-coupled Network	2	1.4	1



(a)



(b)

Fig. 5. The average queue length of loosely coupled network and tightly coupled network with respect to different λ. (a) Loosely coupled network. (b) Tightly coupled network.

In our simulation, we let each region's average load size $\mu = 3$ units, and vary the load arrival rate λ. The average queue length with respect to different λ is plotted in Fig. 5. In the figure, the theoretical bounds are derived by (9). Notice that λ denotes the average number of loads arriving in one unit of time. Since we consider divisible loads which are large in size, it is reasonable to let $\lambda < 1$.

From Fig. 5, we observe that the average queue length increases with λ, which is natural. Further, we notice that the actual average queue length of the tightly coupled network is much smaller than the theoretical bound. However, as the network's communication delay becomes larger, the actual average queue length moves closer to the theoretical bound. This behavior is captured in Fig. 5a. The loosely coupled network's average queue length is quite close to the theoretical bound. Therefore, (9) serves as a very good estimator on average queue length for loosely coupled networks, but may not give a "tight" estimation for tightly coupled networks.

TABLE 2
Regions' Equivalent Computation Capacities for the General Case

$Region_i$	E_i^c	$E_i^{0\prime}$	$E_i^{1\prime}$	$E_i^{2\prime}$	E_i^{max}
$Region_0$	3.3	∞	2.9	2.65	2.5
$Region_1$	4.5	3.5	∞	3.3	2.8
$Region_2$	2.7	2.5	2.4	∞	2.25

Next, we consider a more general case—regions having different computation powers. We generate a network with three regions, and the regions' equivalent computation power in different cases is shown in Table 2, where $E_i^{0\prime}$ denotes the equivalent computation power for $Region_i$ when $Region_0$ is idle, and similar for $E_i^{1\prime}$ and $E_i^{2\prime}$.

Similarly, we let the average load size to be three units, and run the simulation for different sets of load arrival rates. Several results are reported in Table 3, where Q_i^t denotes the theoretical bound of $Region_i$'s average queue length derived from (9) and Q_i^a denotes its actual average queue length. From Table 3, we find that Q_2^a is very close to Q_2^t independent of load arrival rates. This is because $Region_2$ is similar to a loosely coupled network in that its main computation power lies within its critical domain, i.e., it cannot borrow too much extra computation power from other idle regions. However, as regions are able to gain more computation capacity from other idle regions, the difference between Q_i^t and Q_i^a increases. This tendency is shown by $Region_0$ and $Region_1$'s performance. The above phenomena is also intuitive, as these regions' actual average computation capacity is much larger than their critical domains' computation capacity.

From the above discussions, we know that (9) can be directly used as a reference to assign buffer space to regions which exhibit more "loosely coupled" characteristics. However, for the regions exhibiting more "tightly coupled" characteristics, one should reduce the value predicted by (9) correspondingly, and then use the new value as the reference.

7 Conclusions

In this paper, we have addressed the problem of scheduling multisource divisible loads in arbitrary networks. In this study, we considered a very generic graph/network with heterogeneous processing nodes and links. To our knowledge, this is the first attempt to consider scheduling multisource divisible loads on such networks. We had considered each aspect of the problem dimension by analyzing the effects of several key parameters—network size (number of nodes/scalability of the network), rate of arrival of loads, rate of processing of the loads, number of

TABLE 3
Experimental Results for the General Case

	Experiment 1			Experiment 2			Experiment 3		
$Region_i$	λ_i	Q_i^t	Q_i^a	λ_i	Q_i^t	Q_i^a	λ_i	Q_i^t	Q_i^a
$Region_0$	1/15	5.3	3.7	1/14	6.7	4.2	1/17	3.8	2.7
$Region_1$	1/15	26	6.4	1/14	79.9	10.3	1/18	8.2	3.6
$Region_2$	1/15	3.3	2.9	1/18	2.2	2.0	1/13	4.6	4.0

sources, etc. We believe that this study is a very timely contribution to the DLT domain, as it represents a generalization of the problem. One of the key contributions of this study is in the graph partitioning approach and resource sharing across domains. This introduces the possibility of dynamic power tapping of idle resources. We also derived theoretical bounds on the average buffer length at sources when each source has an independent Poisson arrival loads. We summarize all the contributions below.

We proposed a novel GP scheme. GP solves the fundamental problem of scheduling loads in arbitrary networks—that is, from which source and which route a node should receive loads. Further, GP works very efficiently in the sense that it combines network partitioning and spanning tree generation in a single step. Then, based on GP, we proposed two novel scheduling strategies—SSS and DSS. SSS applies to the static case where no new loads will arrive in the network, while DSS to the dynamic case where loads arrive randomly. We also studied the dynamic behavior of DSS using queuing theory, and our analysis revealed the upper bound of each load's average waiting time and each source's average queue length. Both SSS and DSS have shown a "hybrid" property of superposition and network partitioning, and our simulation has verified the effectiveness and usefulness of SSS and DSS. Further, the simulation has revealed a very interesting characteristic of SSS that in loosely coupled networks, an increasing of number of sources will not affect the total processing time significantly when the number of sources is below a threshold. Our simulation also shows that for DSS, the theoretical bound derived for each source's average queue length is very useful to predict the actual average queue length in loosely coupled networks, but it may not be tight for tightly coupled networks.

An immediate extension to this work is to study the dynamic behavior of tightly coupled networks more precisely. Further, a dedicated network is considered in this paper. One may attempt to incorporate the time-varying nature of the speeds of links and processors into problem formulation, and study the multisource scheduling problem in a real dynamic environment.

REFERENCES

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

[2] V. Bharadwaj, D. Ghose, V. Mani, and T.G. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems,* CS Press, 1996.

[3] S.K. Chan, V. Bharadwaj, and D. Ghose, "Large Matrix-Vector Products on Distributed Bus Networks with Communication Delays Using the Divisible Load Paradigm: Performance Analysis and Simulation," *Math. and Computers in Simulation,* vol. 58, pp. 71-79, 2001.

[4] J. Blazewicz, M. Drozdowski, and M. Markiewicz, "Divisible Task Scheduling—Concept and Verification," *Parallel Computing,* vol. 25, pp. 87-98, Jan. 1999.

[5] M. Drozdowski and P. Wolniewicz, "Experiments with Scheduling Divisible Tasks in Clusters of Workstations," *Proc. Euro-Par '00,* pp. 311-319, 2000.

[6] D. Ghose and H.J. Kim, "Computing BLAS Level-2 Operations on Workstation Clusters Using the Divisible Load Paradigm," *Math. and Computer Modelling,* vol. 41, pp. 49-71, Jan. 2005.

[7] M. Drozdowksi and P. Wolniewicz, "Divisible Load Scheduling in Systems with Limited Memory," *Cluster Computing,* special issue on divisible load scheduling, vol. 6, no. 1, pp. 19-30, Jan. 2003.

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

[9] P. Wolniewicz, "Multi-Installment Divisible Job Processing with Communication Startup Cost," *Foundations of Computing and Decision Sciences,* vol. 27, no. 1, pp. 43-57, 2002.

[10] V. Bharadwaj and G. Barlas, "Scheduling Divisible Loads with Processor Release Times and Finite Size Buffer Capacity Constraints," *Cluster Computing,* vol. 6, pp. 63-74, 2003.

[11] Y. Yang, K. van der Raadt, and H. Casanova, "Multiround Algorithms for Scheduling Divisible Loads," *IEEE Trans. Parallel and Distributed Systems,* vol. 16, no. 11, pp. 1092-1102, Nov. 2005.

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

[13] M. Moges and T. Robertazzi, "Grid Scheduling Divisible Loads from Multiple Sources via Linear Programming," *Proc. IASTED Int'l Conf. Parallel and Distributed Computing and Systems,* 2004.

[14] M. Moges, T. Robertazzi, and D. Yu, "Divisible Load Scheduling with Multiple Sources: Closed Form Solutions," *Proc. Conf. Information Sciences and Systems,* The Johns Hopkins Univ. Mar. 2005.

[15] T. Lammie and T. Robertazzi, "A Linear Daisy Chain with Two Divisible Load Sources," *Proc. Conf. Information Sciences and Systems,* The Johns Hopkins Univ., Mar. 2005.

[16] T.G. Robertazzi and D. Yu, "Multi-Source Grid Scheduling for Divisible Loads," *Proc. Conf. Information Sciences and Systems,* pp. 188-191, Mar. 2006.

[17] J.A. Broberg, Z. Liu, C.H. Xia, and L. Zhang, "A Multi-commodity Flow Model for Distributed Stream Processing," *ACM SIGMETRICS Performance Evaluation Rev.,* vol. 34, no. 1, pp. 377-378, 2006.

[18] C.H. Xia, J.A. Broberg, Z. Li, and L. Zhang, "Distributed Resource Allocation in Stream Processing Systems," *Proc. 20th Int'l Symp. Distributed Computing,* pp. 489-504, 2006.

[19] J.T. Hung and T.G. Robertazzi, "Divisible Load Cut through Switching in Sequential Tree Networks," *IEEE Trans. Aerospace and Electronic Systems,* vol. 40, no. 3, pp. 968-982, July 2004.

[20] P. Byrnes and L.A. Miller, "Divisible Load Scheduling in Distributed Computing Environments: Complexity and Algorithms," Technical Report MN ISYE-TR-06-006, Univ. of Minnesota, Graduate Program in Industrial and Systems Eng., 2006.

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

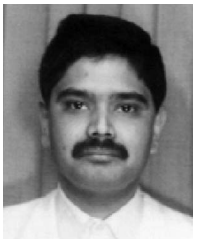
[22] D. England, B. Veeravalli, and J. Weissman, "A Robust Spanning Tree Topology for Data Collection and Dissemination in Distributed Environments," *IEEE Trans. Parallel and Distributed System,* vol. 18, no. 5, pp. 608-620, May 2007.

[23] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. MIT Press, Sept. 2001.

[24] D. Bertsekas and R. Gallager, *Data Networks.* Prentice-Hall, 1992.

[25] J. Little, "A Proof of the Queueing Formula $L = \lambda W$," *Operations Research J.,* vol. 18, pp. 172-174, 1961.

Jingxi Jia received the BEng degree in computer science and engineering from the University of Electronic Science and Technology of China. Currently, he is a PhD candidate of electrical and computer engineering from the National University of Singapore, and he submitted his thesis in 2008. His main research interests are in grid computing, load balancing, and parallel and distributed systems.



Bharadwaj Veeravalli received the BSc degree in physics from Madurai-Kamaraj University, India, in 1987, the master's degree in electrical communication engineering from the Indian Institute of Science, Bangalore, India, in 1991, and the PhD degree from the Department of Aerospace Engineering, Indian Institute of Science, Bangalore, India, in 1994. He did his postdoctoral research in the Department of Computer Science, Concordia University, Montreal, Canada, in 1996. He is currently with the Department of Electrical and Computer Engineering, Communications and Information Engineering (CIE) Division, National University of Singapore, as a tenured associate professor. His mainstream research interests include multiprocessor systems, cluster/grid computing, scheduling in parallel and distributed systems, bioinformatics and computational biology, and multimedia computing. He is one of the earliest researchers in the field of DLT. He has published more than 100 papers in high-quality international journals and conferences and contributed 10 book chapters. He had successfully secured several externally funded projects. He has coauthored three research monographs in the areas of PDS, distributed databases (competitive algorithms), and networked multimedia systems, in the years 1996, 2003, and 2005, respectively. He had guest edited a special issue on cluster/grid computing for IJCA, USA journal in 2004. He is currently serving the editorial boards of the *IEEE Transactions on Computers*, *IEEE Transactions on SMC-A*, and *International Journal of Computers and Applications*, US, as an associate editor. He had served as a program committee member and as a session chair in several international conferences. His academic profile, professional activities, mainstream and peripheral research interests, research projects and collaborations, and most recent list of publications can be found on http://cnds.ece.edu.sg/elebv. He is a senior member of the IEEE and the IEEE Computer Society.

Jon Weissman received the BS degree from Carnegie-Mellon University in 1984, and the MS and PhD degrees from the University of Virginia in 1989 and 1995, respectively, in computer science. He is a leading researcher in the area of high-performance distributed computing. His involvement dates back to the influential Legion project at the University of Virginia during his PhD. He is currently an associate professor of computer science at the University of Minnesota where he leads the Distributed Computing Systems Group. His current research interests are in grid computing, distributed systems, high-performance computing, resource management, reliability, and e-science applications. He works primarily at the boundary between applications and systems. He is a senior member of the IEEE and the IEEE Computer Society, and awardee of the US National Science Foundation (NSF) CAREER Award (1995).