Requirement-Aware Strategies with Arbitrary Processor Release Times for Scheduling Multiple Divisible Loads

Menglan Hu and Bharadwaj Veeravalli, Senior Member, IEEE

Abstract—This paper investigates the problem of scheduling multiple divisible loads in networked computer systems with a particular emphasis in capturing two important real-life constraints, the arbitrary processor release times (or ready times) and heterogeneous processing requirements of different loads. We study two distinct cases of interest, static case, where processors’ release times are predetermined and known, and dynamic case, where release times are unknown until processors are released. To address the two cases, we propose two novel scheduling strategies, referred to as Static Scheduling Strategy (SSS) and Dynamic Scheduling Strategy (DSS), respectively. In addition, we capture a task’s processing requirements in our strategies, a unique feature that is applicable for handling loads on networks that run proprietary applications only on certain nodes. Thus, each task can only be processed by some certain nodes in our formulation. To handle the contention of multiple applications that have various processing requirements but share the same processing nodes, we propose an efficient load selection policy, referred to as Most Remaining Load First (MRF). We integrate MRF into SSS and DSS to address the problem of scheduling multiple divisible loads with arbitrary processor release times and heterogeneous requirements. We evaluate the strategies using extensive simulation experiments.

Index Terms—Divisible loads, parallel processing, communication delay, processing time, multiple applications, release times.

1 INTRODUCTION

Scheduling tasks in distributed systems is a critical issue for achieving high performance. One type of scheduling problems is one in which no dependencies exist among tasks, and the number and sizes of tasks can be arbitrarily chosen. In practice, this is the case for various large-scale applications in scientific domains such as signal processing, image processing, and database searching. Such applications are structured as large numbers of independent, identical, and low-granularity computations and are thus amenable to straightforward parallel computing. These applications have been called divisible loads because a scheduler may arbitrarily divide the loads among networked computers, in terms of both the numbers and sizes of tasks. The above scheduling problem can be characterized using the divisible load model, which has been studied extensively in the last two decades, resulting in a cohesive theory called Divisible Load Theory (DLT) [1]. DLT offers a tractable and powerful tool to scheduling that allows linear and continuous modeling of partitionable computations and communications for parallel processing. A vast literature on DLT demonstrates it as an elegant methodology for handling large-scale applications in distributed systems [2], [3], [4], [5], [6]. In practice, this divisible load model is an approximation of an application consisting of large numbers of identical, independent, and low-granularity computations, and has been applied to a wide spectrum of real-life applications including database searching [7], matrix computing [8], and biological sequencing [9].

Prior DLT works usually assume that all processors are simultaneously available for a task. However, in real-life application execution some processors may have been allocated to previously admitted tasks. Such processors will not be available until their previously admitted tasks are completed. Thus, it is worth studying the case that processors have different release times (or ready times). A few prior works have studied the problem of scheduling divisible loads with different processor release times. Prior work [11] presented strategies for scheduling divisible loads on bus networks with arbitrary processor ready times. Closed-form solutions were derived for identical release times and a heuristic was presented for arbitrary release times. Another work [10] studied scheduling divisible loads on linear chain networks with arbitrary processor ready times. In [10], both single and multi-installment strategies were presented for the cases that a minimum processing time can be achieved. Two heuristics were proposed for the cases that a minimum processing time cannot be realized. Chuprat and Baruah [12] investigated scheduling real-time divisible loads with arbitrary processor release times on clusters. In these works, predetermined and known ready times are assumed. However, in real-life scenarios, this assumption is not always satisfied. In many cases, processors’ release times are indeterminable or unknown until they become available. Therefore, it is beneficial to consider the case of dynamic processor ready times, where ready times are unknown until processors are released. In this study, we aim to study both static and dynamic processor release times.
Also, the problem of scheduling multiple divisible loads received only limited attention due to its complexity. Marchal et al. [15] proposed a novel network model for wide area networks and proposed heuristics for seeking suboptimal solutions for a steady-state multitask divisible load scheduling problem. Drozdowski et al. [13] showed that scheduling multiple divisible loads on star networks is computationally hard and hence they identified special cases solvable in polynomial time. Drozdowski and Lawenda [14] showed that scheduling multiple divisible loads on homogeneous star networks is again computationally hard. They also presented polynomial time solutions for special cases. Notice that in these works, processing nodes are assumed to be capable of processing all divisible load applications. However, in real-life scenarios, this assumption does not always hold true. In practice, due to variations in different types of applications and processing nodes, not all nodes may be capable of computing all types of applications. A typical example is that there may exist proprietary applications that are licensed and available only at specific nodes. Also, processing nodes’ hardware may become bottlenecks to restrict their capabilities of processing various applications. For example, an application requiring 1 GB RAM should be assigned to nodes that have at least that much RAM. Therefore, it is desirable to investigate the scheduling of multiple divisible loads with heterogeneous processing requirements. We refer to this context simply as requirement-aware scheduling problem.

Thus, the above discussion clearly sets the motivation for the problem addressed in this paper. This study differs to the prior works [10], [11], [12], [13], [14], [15] in two novel ways. First, this work considers both static and dynamic processor release times while in the prior works processors’ ready times are assumed to be predetermined before execution starts. Second, this paper considers multiple loads with heterogeneous processing requirements, which is not considered in most prior DLT papers. To the best of our knowledge, this study is the first attempt of its kind to consider the combined influence of different release times and heterogeneous processing requirements. This paper contributes two novel scheduling strategies, referred to as Static Scheduling Strategy (SSS) and Dynamic Scheduling Strategy (DSS). These strategies provide efficient load balancing and resource utilization while elegantly addressing arbitrary release times. Also, we present a requirement-aware load selection policy, referred to as Most Remaining Load First (MRF), and integrate it into SSS and DSS to handle the contention among different loads with various processing requirements.

The rest of the paper is organized as follows: Section 2 introduces mathematical models, assumptions, and problem formulation. Section 3 considers static release times and presents SSS and MRF. Section 4 investigates dynamic release times and proposes DSS. In Section 5, we present simulation results to evaluate the proposed algorithms. Finally, Section 6 concludes the paper.

2 Problem Formulation

Our target computing platform is a cluster system connected through a local area network (LAN). The input data are divisible loads originally residing in a single machine, \( P_0 \), which is the master node for scheduling the loads. From \( P_0 \)'s perspective, the logical topology of the platform is actually a star topology. This star topology is the generic architecture for implementing master-worker computations and widely used by prior works including [3], [4], [5], [6]. Let \( M \) be the number of processing nodes in the system and the nodes are denoted as \((P_1, P_2, \ldots, P_M)\). \( P_0 \) does not participate in processing but dispatches loads to the \( M \) nodes for parallel processing. The ready time of processor \( P_i \) \((i = 1, 2, \ldots, M)\) is \( T_i^R \). We consider both static and dynamic processor ready times. We assume that in static case ready times are predetermined and known before the execution starts; in dynamic case a processor’s ready time is unknown until it is released. Once a processor is released, it can be fully utilized by \( P_0 \).

We consider \( J \) divisible load applications, \( L_j, 1 \leq j \leq J \). The goal of our research is to minimize the total processing time of all the loads. These divisible loads consist of large numbers of identical, independent, and low-granularity data units which can be distributed to networked computers for parallel processing. Also, the divisible loads may be of different types. For instance, one application may deal with files and another with matrices. Following Marchal et al.’s model [15], we have some definitions below. We partition each load into a large number of identical, independent, and low-granularity load units (a file, or a matrix) and let \( N_j \) be the total number of units of load \( L_j \). Also, we let \( C_j \) be the amount of computation required to process a load unit for \( L_j \) and \( S_j \) be the size (in bytes) of a load unit for \( L_j \). The parameters \( C_j \) and \( S_j \) represent the task granularity of load \( L_j \). We can obtain \( C_j \) by testing one data unit of the load \( L_j \) on a single processor and thus acquire the exact amount of computations of \( L_j \) as \( N_j \) and \( S_j \) are known [9]. Furthermore, as we consider different types of applications and heterogeneous processing nodes, these divisible loads have different computing requirements and each load can only be processed by some certain processors. To guarantee that all loads can be successfully completed, each load can be processed by at least one processor.

In this paper, start-up time overheads are considered to be negligible and we adopt a linear and fixed communication and computation model, which is widely used in many prior DLT studies including [2], [6], [15]. In this linear model, the communication speed (bandwidth) of the LAN is \( B \). It takes \( \frac{X_j}{B} \) time units to send \( X \) units of load \( L_j \) from \( P_0 \) to each processor \( P_i \). Similarly, each processor \( P_i \) has a compute power \( W_i \). It takes \( \frac{X_j}{W_i} \) time units to compute \( X \) units of \( L_j \) on \( P_i \). In addition, we assume that \( B \) and \( W_i \)'s are fixed as the cluster is a dedicated system for the master node \( P_0 \) to execute divisible load applications. We will discuss why one may consider removing the assumption of fixed communication and computation speeds in Section 6.

We assume that \( P_0 \) sends loads in a sequential mode: loads are not sent to the processors simultaneously. This is the common assumption in DLT literature [2], [3], [4], and is justified by the behavior of LAN. It may be noted that current technologies allow simultaneous load transmission to multiple processors, but this parallel communication mode cannot significantly improve the performance of our strategies for two reasons. First, the overall communication
throughput of $P_0$ is constrained by the bandwidth of the LAN, not the number of receivers. Second, as our strategies have already guaranteed efficient overlapping of communications with computations, there is very little room for the parallel mode to improve the performance. Therefore, we adopt the sequential mode as the parallel mode can cause unnecessary complexities for scheduling. Notice that in some other scenarios the parallel mode may significantly outperform the sequential mode and we will discuss this in Section 6. Finally, we assume that each processor has adequate storage to store and compute any amount of data. When compared to the time taken for computation and communication, the time for reporting the result back to $P_0$ is negligible. The time taken for making scheduling decisions is also negligible.

3 Static Scheduling Strategy

In this section, we consider scheduling multiple divisible loads in the case of static processor release times and present Static Scheduling Strategy. Fig. 1 shows the timing diagram of SSS for an example of four processors and one load. The identifier $\alpha_{ik}$ indicates the load chunk allocated to processor $P_i$ in round $k$. As shown in Fig. 1, SSS is a phase-based multiround approach. SSS works in an incremental fashion, consuming several rounds for scheduling and computing loads. In each round, SSS works by iterating the steps of selecting a load, selecting a processor, and then allocating a load chunk from the load to the processor. The loads distributed in each round is computed in a corresponding phase. The length of each phase is $T_p$, which is fixed and predetermined. Below, we describe the detailed design of SSS and we consider an arbitrary round $k (k = 1, 2, 3, \ldots)$ as an example to illustrate the working flow of SSS.

As multiple heterogeneous loads are involved, $P_0$ needs to first select a load before it can allocate a load chunk to some processor. The key issue of load selection is to address the contention of various loads which share the same compute resources but have various processing requirements. To design an efficient load selection policy, we need to achieve high processor utilization in scheduling. If a processor is only able to compute certain loads but all those loads have been completed, it will retain idle and this naturally wastes its compute power. Also, we observe that such wastes severely happen at the end of the entire execution as many loads gradually get completed and hence many processors will remain idle. Specifically, if the processing of a load completes much later than others, many processors that cannot compute it have to retain idle for long time, resulting in serious compute power waste. Therefore, to minimize this waste and achieve high performance, we should provide fairness to different loads and prevent some loads from being completed much later than other loads. According to this principle, we propose a fair load selection policy, referred to as Most Remaining Load First. This policy selects the load with the most remaining amounts (in terms of required computations). We denote $L_j^R$ as the total amount of computations required for processing all unallocated units of $L_j$. We can write $L_j^R$ as:

$$L_j^R = N_j^R C_j,$$

where $N_j^R$ is the number of unallocated units of $L_j$. Thereby, among all available loads, MRF selects the one with the largest $L_j^R$. Since MRF first selects the loads with the most remaining amounts, it can effectively prevent some loads from being completed much later than other loads. Notice that at the beginning of each round all remaining loads are set as available. If $P_0$ cannot find any processor for a selected load in round $k$, then this load will be set as unavailable.

Suppose load $L_j$ is selected for scheduling. Now $P_0$ needs to select a processor and allocate it a load chunk from $L_j$. To make efficient scheduling decisions, we should first consider the workings of the processors to understand the rationale behind the scheduling. In SSS, the processors compute chunks once they have any unprocessed chunks. Also, they process chunks in a first-come-first-serve way. Hence, following a common strategy, in each round $k$, $P_0$ attempts to dispatch appropriate amounts of loads to processors so that the processors can finish their chunks of round $k$ simultaneously when phase $(k + 1)$ ends (as shown in Fig. 1). The strategy that guarantees processors to finish simultaneously is widely used in many DLT works [1] since this strategy achieves load balancing and optimizes resource utilization.

For processor selection in round $k$, $P_0$ only considers the processors satisfying 3 criteria: first, such processors are capable of processing $L_j$. Second, such processors are released before phase $(k + 1)$ ends because all processors are scheduled to finish the chunks of round $k$ when phase $(k + 1)$ ends to conform to the above rule. Third, such processors have idle time before phase $(k + 1)$ ends. If a released processor is allocated an adequate amount of load to continually compute until phase $(k + 1)$ ends, the processor is categorized as having no idle time before phase $(k + 1)$ ends; otherwise, it is classified as having idle time before phase $(k + 1)$ ends.

The processors satisfying the 3 criteria are considered for $L_j$ and $P_0$ picks one among them to allocate a chunk. Since, all processors have the same communication speeds and their compute speeds are independent to loads, thus $P_0$ randomly selects a processor among all available ones. If no processor is available for $L_j$, $L_j$ will be set as unavailable for round $k$. This means $L_j$ cannot be scheduled in this round but it will be considered in next rounds. Then $P_0$ executes...
MRF again to select another load. If no loads are available, \( P_0 \) will end round \( k \). It may be noted that two cases can cause that no loads are available. The first is that all loads have been finished allocation. The second is that all released processors that can compute the loads have been allocated enough load chunks so that they have no idle time. In this case, \( P_0 \) will immediately start round \( (k + 1) \).

There is another rule for ending round \( k \) as explained below. The transmission of round \( k \) must be finished before phase \( (k + 1) \) starts so that processors can start immediately to compute the chunks of round \( k \) in phase \( (k + 1) \) and \( P_0 \) can also start immediately to transmit the chunks of round \( (k + 1) \) in phase \( (k + 1) \). Thus, when phase \( k \) ends \( P_0 \) will immediately terminate the communication of round \( k \) and start round \( (k + 1) \). Therefore, if there are too many processors so that \( P_0 \) cannot utilize all of them, then \( P_0 \) will abandon some processors, guaranteeing that the communication of round \( k \) can be finished before phase \( (k + 1) \). This rule also guarantees the loads with the most remaining amounts can be timely scheduled in the next round.

Suppose \( P_0 \) has selected load \( L_j \) and processor \( P_i \) for scheduling, now a load chunk can be allocated from \( L_j \) to \( P_i \).

In round \( k \), \( P_i \) dispatches appropriate amounts of load to \( P_i \) so that the chunk can be guaranteed to reach \( P_i \) before phase \( (k + 1) \) starts and \( P_i \) can finish the chunk when phase \( (k + 1) \) ends (as shown in Fig. 1). To allocate a chunk, \( P_i \) first calculates \( P_i \)'s available compute time before phase \( (k + 1) \) and \( P_i \) then can allocate a chunk according to the available time. We denote this time as \( T_{ik \text{comp}} \). To obtain \( T_{ik \text{comp}} \) we should consider four constraints. First, \( P_i \) can process loads only after it becomes ready. Second, \( P_i \) can process the new chunk only after it finishes previously allocated chunks. Third, even if \( P_i \) is ready and idle, necessary communication should be subtracted from the total available time. Finally, the communication of \( P_i \) should be finished before phase \((k + 1)\) starts. We can obtain \( T_{ik \text{comp}} \) as follows:

\[
\max T_{ik \text{comp}},
\]

subject to

\[
\begin{align*}
T_{ik \text{comp}} &\leq T_p(k+1) - T_p^R, \\
T_{ik} &\equiv T_p(k+1) - T_p^F, \\
T_{ik \text{comp}} &\leq T_p(k+1) - T_{\text{comm}}, \\
T_{ik \text{comp}} &\leq \frac{C_j}{W_{jS_i}},
\end{align*}
\]

where \( T_p^F \) is \( P_i \)'s finish time of its previously allocated chunks, which is obtained according to prior load allocation. \( T_{\text{comm}} \) is the time instant when \( P_0 \) is allocating loads to \( P_i \) and \( T_p(k+1) - T_{\text{comm}} \) is the total available time when \( P_0 \) is allocating loads to \( P_i \). The above linear program can be simplified as follows:

\[
T_{ik \text{comp}} = \min(E_1, E_2, E_3, E_4),
\]

where

\[
\begin{align*}
E_1 &= T_p(k+1) - T_p^R, \\
E_2 &= T_p(k+1) - T_p^F, \\
E_3 &= \frac{T_p(k+1) - T_{\text{comm}}}{1+\frac{T_p}{T_p^R}}, \\
E_4 &= \frac{BC_j(kT_p - T_{\text{comm}})}{W_{jS_i}}.
\end{align*}
\]

As \( E_1, E_2, E_3, \) and \( E_4 \) in (5) can be directly calculated, we can easily determine \( T_{ik \text{comp}} \) and obtain the chunk size \( \text{Chunk} \) (in terms of number of units) allocated to \( P_i \) from \( L_j \).

In reality, this number should be an integer. Thus, we can write \( \text{Chunk} \) as follows:

\[
\text{Chunk} = \left\lfloor \frac{W_{jS_i}T_{ik \text{comp}}}{C_j} \right\rfloor.
\]

If \( N^R \) is less than the chunk size \( \text{Chunk} \), then \( \text{Chunk} \) is accordingly reduced to that number and all unallocated units of \( L_j \) are allocated to \( P_i \). In this case, \( P_i \) will be available after allocated the chunk since it still has some idle time before phase \((k + 1)\) ends. After dispatching the chunk, \( P_0 \) iterates the above steps of load selection, processor selection, and load allocation until no loads are available for this round, or phase \( k \) terminates. Then \( P_0 \) terminates round \( k \) and starts round \((k + 1)\), repeating scheduling loads until all the loads are allocated.

Now, we discuss on the mechanism of SSS and highlight its advantages. From Fig. 1 we can observe that, in phase \((k + 1)\), while computing the chunks of round \( k \), processors can simultaneously receive the chunks of incoming rounds. In this way, communications and computations are pipelined and overlapped to save time. Prior DLT works usually adopt a single-round approach, in which processors need to wait for \( P_0 \) (thus, wasting time) to transmit loads to other processors before they can start their own communication and computation [4], [6]. But in SSS such waste is significantly reduced by efficient overlapping. In addition, in each round \( P_0 \) dispatches loads according to processors’ available time so that all processors can simultaneously finish their loads of that round. This naturally achieves load balancing in each round. Further, SSS offers efficient resource utilization since processors can continuously work throughout consecutive phases, provided processors can process enough numbers of loads.

It may be noted that in the last round, processors may finish computing at different times. But this influences the overall performance little because the unbalancing is limited in one phase and for all other rounds efficient load balancing is guaranteed. However, if the number of rounds is too small, for example, less than five, unbalanced amounts of load in the last round may become considerable.

Therefore, the length of each phase \( T_p \) is not arbitrarily chosen. If \( T_p \) is too large, unbalanced load in the final round will become significant and seriously weakens the performance. In addition, the time taken in waiting for communication in round 1 will also become considerable. On the other hand, \( T_p \) cannot be too small because the data units cannot be infinitesimally small. Therefore, we should avoid determining \( T_p \) as either very small or very large. When implementing SSS in real-world applications, we should check the task granularity and the total size of the data before \( T_p \) can be determined. In Section 5, we will evaluate the impact of \( T_p \) on our strategies.

4 Dynamic Scheduling Strategy

Now, we tackle a more realistic situation wherein processors that will eventually participate are unknown until they
become available. To handle unexpected processor releases, we revisit SSS to carry out an alternate design, namely Dynamic Scheduling Strategy. DSS is again a phase-based multiround approach. Fig. 2 is a timing diagram of DSS showing an example of four processors and one load. The identifier $\alpha_{ik}$ means the chunk allocated to processor $P_i$ in round $k$. Same as SSS, in each round, DSS iterates the steps of selecting a load by MRF, selecting a processor, and then allocating a load chunk from the load to the processor according to (4) and (6). These steps repeat until all loads are completed. Since the steps have been thoroughly described in Section 3, we omit to detail these again, but below we intend to describe the differences between SSS and DSS.

In SSS, load distribution can be determined before the entire execution, but in DSS the distribution should be dynamically generated in each round since a processors’ participation is unknown until it is released. In other words, we should avoid making scheduling decisions too early since such decisions may result in poor utilization of new processors. Therefore, DSS works in a periodic fashion: for each round $k$, $P_0$ starts dispatching at the beginning of phase $k$ and distributes the chunks of this round in phase $k$. Thus, the chunks sent in phase $k$ can timely reach the processors before the computation of round $k$ starts in phase $(k + 1)$. Same as SSS, in DSS the transmission of all chunks of round $k$ must be finished before phase $(k + 1)$ starts so that processors can timely start to compute these chunks in phase $(k + 1)$ and $P_0$ can also timely start to transmit the chunks of round $(k + 1)$ in phase $(k + 1)$. When phase $k$ ends $P_0$ terminates round $k$ and immediately starts the next round. Therefore, the communication time of round $k$ is only limited in phase $k$ (as shown in Fig. 2).

Notice that $P_0$ may finish the communication of round $k$ before phase $k$ ends. But $P_0$ will not start round $(k + 1)$ until phase $(k + 1)$ starts. This is because a processor’s ready time is unknown until it is released; if load distribution is determined too early and loads are sent too early, when new processors are released the loads may have been allocated to early-released processors and the new processors may not be involved in computation. Although, it is possible to redistribute loads to the new processors and the early-released processors so that the new processors can also be utilized, this process will waste communication resources, and become cumbersome to handle scheduling and load balancing, especially in the presence of heterogeneous loads. Therefore, in DSS $P_0$ always starts the communication of round $k$ at the beginning of phase $k$ and works in a periodic way as we mentioned before. After sending loads to all available processors, $P_0$ will retain idle, waiting until phase $k$ ends, and then start a next round.

DSS handles newly released processors in a different way to SSS. In fact, in SSS new processors are not explicitly addressed, but in DSS extra efforts are needed to handle dynamic processor releases. In DSS, $P_0$ keeps monitoring new releases. If a new processor $P_j$ becomes ready in phase $k$, $P_0$ will realize this immediately. If $P_0$ is busy when $P_j$ becomes ready, $P_0$ will handle it after the current transmission; otherwise, $P_0$ will immediately address it. To handle the new processor, $P_0$ checks all the loads that can be processed by $P_j$. If some loads that can be executed by $P_j$ are set as unavailable but they have unallocated units, $P_0$ will set such loads as available. Thus, these loads can be considered again in this round and $P_j$ can also be utilized immediately. Then $P_0$ continues the steps of scheduling loads.

Now, we conclude the mechanisms and benefits of DSS. In order to handle new processors dynamically, DSS works in a periodic fashion so that scheduling decisions will not be made too early. Also, DSS elegantly handles unexpected release times by dynamically scheduling loads to new processors once they become ready. Moreover, same as SSS, DSS provides efficient load balancing and resource utilization as we analyzed in Section 3. DSS also implements efficient pipelining and overlapping of communication with computation (as shown in Fig. 2) to optimize processing time.

5 Performance Evaluation

In this section, we evaluate the performance of SSS and DSS strategies by rigorous simulation experiments. We typically follow the style of simulation study used in most earlier studies in DLT literature [2], [4], [6].

To evaluate SSS and DSS, we compare their performance with an ideal case (the performance bound) as there are no strategies available in the literature to compare in this current problem context directly. For the ideal case, we assume that arbitrary processor releases are perfectly handled so that each processor is immediately involved in computation once it is released. Also, we assume that the ideal case is not influenced by the granularity of the loads, which means the number of load units can be rational numbers when simulating the ideal case. In addition, in the ideal case communications can be “optimally” overlapped with computation such that communication delays are zero. However, if the network bandwidth is too small, communications may not absolutely hidden even if an “optimal” scheduling policy is applied. Hence, we recognize a performance bound for the communication of the ideal case. That is, in the ideal case, the total processing time of $X$ units of load $L$ cannot be less than the minimum communication time for transmitting such a load chunk, which is $\frac{T}{P}$ seconds. If a calculated processing time of the ideal case is less than its minimum communication time, we let the processing time be the minimum communication time.
Therefore, the ideal case presents the performance bound among all possible solutions since the resource utilization of either computation or communication is 100 percent in this case. In the following figures, the results of the ideal case are denoted as “Ideal.”

The normal experiment sets are denoted as “SSS-MRF” and “DSS-MRF.” Also, we simulate an ideal load allocation case (ILA), in which each processor can compute all loads. We apply ILA to both SSS and DSS to produce “SSS-ILA” and “DSS-ILA,” respectively. Thus, by comparing SSS/DSS-MRF with SSS/DSS-ILA, we can study the effect of heterogeneous loads and evaluate MRF. By comparing SSS/DSS-ILA with the ideal case, we can evaluate the effectiveness of SSS/DSS in handling homogeneous loads.

The simulated system consists of randomly generated processing nodes. The initial configurations are set as follows: the number of processing nodes $M$ is 20 and the number of loads $J$ is 10. For each processor $P_i$, the computation power $W_i$ is uniformly distributed among $[100, 500]$ and the communication speed $B$ is set as 100. Processors’ ready times are uniformly distributed among $[0, 100]$ time units. As stated before, for SSS the processors’ ready times are known before execution starts and for DSS the processors’ ready times are unknown until they are released. In addition, to study the influence of different load types and sizes, we let $S_j$ be uniformly distributed among $[0.5, 1.5]$, let $C_j$ be uniformly distributed among $[1, 100]$, and let $N_j$ be uniformly distributed among $[500, 1500]$. The length of each phase $T_p$ is 10 time units. To study the influence of multiple loads with heterogeneous requirements, we use parameter $p$ to denote the probability that a load $L_j$ can be processed by any processor. We let $p_j$ be uniformly distributed among $[0.2, 0.8]$.

In the following experiments, we vary our interested parameters while fixing other parameters as their initial values to study the effect of the interested parameters. We first vary $B$ from 20 to 200 and the corresponding processing time is shown in Fig. 3a. In addition, we vary $M$ from 4 to 40 and Fig. 3b plots the processing time versus $M$. Further, we vary $J$ from 2 to 20 and Fig. 3c depicts the processing time versus $J$. Then, to study the effect of $p_j$, we let $p_j$ be uniformly distributed among $[p_a - 0.1, p_a + 0.1]$ and we vary $p_a$ among $[0.2, 0.8]$, and the corresponding processing time is shown in Fig. 3d. Moreover, to investigate the effect of different load groups with different $p$ values, we consider a load set consisting of two groups—group 1 are “easy” loads for which $p_j$ is uniformly distributed among $[0.5, 0.9]$, and group 2 are “hard” loads for which $p_j$ is uniformly distributed among $[0.1, 0.5]$. We use $p_b$ to denote the proportion of the number of loads in group 1 to the number of the loads in the whole load set. We vary $p_b$ among $[0, 1]$ and Fig. 3e plots the processing time versus $p_b$. Finally, we vary $T_p$ from 0.3125 to 160 and the corresponding processing time is shown in Fig. 3f.
higher resource utilization in SSS than in DSS. Moreover, the available communication time of each round is limited to one phase in DSS, but may be more than one phase in SSS. Accordingly, in each round $P_0$ can send more loads in SSS than in DSS.

Fig. 3a shows that the processing time of all experiment sets stabilizes when $B$ is greater than 80. This is due to the fact that when the network bandwidth is large enough communication can be efficiently overlapped with computation and hence varying $B$ impact little on the performance. In the contrary, when $B$ is smaller than 80, the processing time sharply increases as $B$ decreases since when $B$ is small, communication requires too much time and thus cannot be effectively hidden. Moreover, when $B$ is large the performance gap between SSS and DSS becomes small. A possible reason is that since the advantages of SSS to DSS lie in communications, the advantages will become small when communication speeds are high enough. Similarly, when $B$ is small all experiment sets deliver close performance, indicating that for slow networks communication delays dominate the performance and other factors become negligible.

Fig. 3b shows that as $M$ increases the processing time of all sets decreases as more nodes computing in parallel saves more time. But the performance of SSS and DSS gradually stabilizes when $M$ is greater than 30. A plausible explanation is that when there are abundant processors, network bandwidth becomes a bottleneck and restricts resource utilization. Further, the performance gap between SSS/DSS-ILA and SSS/DSS-MRF is significant when $M$ is small, but gradually decreases as $M$ grows. This indicates that if there are only a few processing nodes, MRF cannot always find proper nodes for scheduling, but if there are abundant nodes probably MRF can find proper nodes.

From Fig. 3c, we observe that the performance gap between SSS/DSS-MRF and the ideal case gradually decreases as $J$ grows. One contributing factor is that the required computation time is approximately proportional to the total load size but the communication delays which cannot be overlapped are merely related to $T_p$, which is fixed. Thus, the effect of communication delays become more significant when the processing time shortens as $J$ decreases. Another contributing factor is that when there are a few loads, the choices for MRF are very limited. Thus, MRF may not be as efficient as ILA. This also explains the fact that the performance gap between ILA and MRF decreases as $J$ grows.

From Fig. 3d, we observe that when $p_a$ is less than 0.4 the processing time of SSS/DSS-MRF sharply increases as $p$ decreases. A plausible explanation is that when $p_a$ is less than 0.4, a processor is unable to compute enough number of loads and thus cannot be fully utilized. Thereby, increasing $p_a$ enables the processors to compute more loads and thus improves the performance. On the other hand, when $p_a$ is greater than 0.4 their performance stabilizes as $p_a$ grows. This is because when $p_a$ is greater than 0.4, a processor becomes efficient to process enough number of loads and thus be fully utilized in computation. Accordingly, increasing $p_a$ impacts little on the performance. Fig. 3e shows another case wherein processors cannot compute enough loads for group 2. Hence, increasing $p_b$ enables the processors to compute more loads and thus improves the performance.

Fig. 3f shows that when $T_p$ is greater than 40, the processing time of SSS/DSS sharply increases as $T_p$ increases. This is due to the fact that the time taken in waiting for communication in round 1 becomes considerable when there are only a few rounds. Also, when $T_p$ is too large, too large load unbalancing in the final round becomes considerable and seriously weakens the performance. In the contrary, when $T_p$ is less than 1.25, the processing time of SSS/DSS significantly increases as $T_p$ decreases. This is because load units cannot be infinitely small. If $T_p$ is too small probably some processors cannot receive a unit as they cannot finish 1 unit in 1 phase. Further, the performance of SSS/DSS stabilizes when $T_p$ is among $[1.25, 40]$. This shows that although $T_p$ cannot be too small or too large, there is still a wide range in which varying $T_p$ only slightly influence the performance of SSS/DSS. This range is related to the granularity and the sizes of the loads. As SSS and DSS are designed for processing divisible loads consisting of large numbers of low-granularity computations, choosing satisfactory values of $T_p$ may be not challenging. When implementing SSS/DSS for real-world applications, simulations or experiments can help determine satisfactory values of $T_p$ once the granularity and size of the data are given.

6 Conclusions

In this paper, we have addressed an important problem of scheduling multiple divisible loads with arbitrary processor release times and heterogeneous processing requirements on networked computing systems. We have proposed two novel scheduling strategies: SSS and DSS. These strategies provide efficient load balancing and resource utilization while elegantly addressing arbitrary release times. In addition, we have proposed a requirement-aware load selection policy (MRF) as described in Section 3 and integrated it into SSS and DSS to handle the contention among different loads with various processing requirements. The simulation results have shown that our strategies are very efficient for both cases under various system parameters.

One future direction is to allow $P_0$ to perform simultaneous communications to processors. The parallel communication mode can be beneficial for computing platforms over wide area networks (WAN) [15] to achieve higher throughput than the sequential communication mode due to bandwidth-sharing properties. In addition, computing platforms on WAN are likely to be nondedicated. Thus, the communication and computation capacities of processors in such systems may fluctuate with time. As computer grids are widely deployed over WAN to perform distributed computing, it is valuable to explore scheduling strategies accommodating the parallel communication mode [15] and fluctuated communication and computation capacities.

Moreover, one may attempt to apply an affine cost model in DLT to include start-up costs. Our strategies can be easily modified for the affine model without affecting the flow of the main content. Suppose $O_{\text{comp}}$ and $O_{\text{comm}}$ are computation and communication start-up overheads. The only
change is to use \( T_p(k+1) - O_{comp} \) to replace \( T_p(k+1) \) and use \( T^{comp} + O_{comp} \) to replace \( T^{comp} \) in (3) and (5). However, in practice, start-up overheads are shown to be small, usually negligible. In [5], such overheads are usually bounded in a few seconds in real-life measurements. When the number of rounds is not too large start-up overheads are indeed negligible. On the other hand, as we demonstrated via simulations, the number of rounds for our strategies should not be large for efficient resource utilization. Thus, the inclusion of affine model may not be a worthwhile attempt. However, as a future work, it would be interesting to determine an optimal, if not, an acceptable number of rounds to be used for maximizing resource utilization.

REFERENCES


