Multi-Round Algorithms for Scheduling Divisible Loads

Yang Yang[1] Henri Casanova [1,2]

[1] Dep. of Computer Science and Engineering [2] San Diego Supercomputer Center

University of California San Diego

Abstract

Divisible load applications occur in many fields of science and engineering, can be easily parallelized in a master-worker fashion, but pose several scheduling challenges. While a number of approaches have been proposed that allocate work to workers in a single round, using multiple rounds improves overlap of computation with communication. Unfortunately, multi-round algorithms are difficult to analyze and have thus received only limited attention. In this paper we answer three open questions in the multi-round divisible load scheduling area: (i) How to account for latencies? (ii) How to account for heterogeneous platforms; and (iii) How many rounds should be used? To answer (i), we derive the first closed-form optimal schedule for a homogeneous platform with both computation and communication latencies, for a given number of rounds. To answer (ii) and (iii), we present a novel algorithm, UMR. We use simulation to evaluate UMR in a variety of realistic scenarios.

Keywords: Parallel Processing, Scheduling, Divisible Loads, Multi-Round Algorithms.

1 Introduction

Assigning the tasks of a parallel application to distributed computing resources to minimize application execution time, or *makespan*, has been studied for a variety of application models, such as the well-known directed acyclic task graph model. Another popular application model is that of independent tasks with no task synchronizations and no inter-task communications, which, albeit simple, arises in most fields of science and engineering. A possible model for independent tasks is one for which the number of tasks and the task sizes, i.e. their computational costs, are set in advance. In this case, the scheduling problem is akin to bin-packing and many heuristics have been proposed in the literature (see [1] for a survey). Another flavor of the independent tasks model is one in which the number of tasks and the task sizes can be chosen arbitrarily. This corresponds to the case when the application consists of an amount of computation, or *load*, that can be arbitrarily divided into any number of independent pieces, or *chunks*. In practice, this model is an approximation of an application that consists of large numbers of identical, low-granularity units of work. This *divisible load* model arises in practice in many domains [2, 3, 4, 5, 6, 7, 8, 9, 6] and has been widely studied in the last several years [10, 11].

Divisible load applications are amenable to straightforward master-worker computing and can thus be easily deployed on computing platforms ranging from commodity clusters to computational grids. However, efficient scheduling is challenging because of the overhead involved when starting tasks. This overhead is due to: (i) the time to transfer application input/output data to/from each compute resource; and (ii) the potential latencies involved when initiating a computation or a communication. The scheduling problem is difficult when the application is neither fully computation-intensive, nor fully communication-intensive. There are two types of approaches for divisible load scheduling. First, one can divide the load in as many chunks as the number of processors, and dispatch them in a *single round* of work allocation. This scheme is simple to design and implement, but leads to poor overlap of communication and computation. The second possibility is to dispatch the load in *multiple rounds*; in each round each worker is allocated a chunk of the load. While single-round approaches have been studied thoroughly (see [11] for a survey), multi-round algorithms are significantly more difficult to analyze and fewer results are available.

The seminal “Multi-Installment” (MI) algorithm [12] is the first (and to the best of our knowledge the only) multi-round algorithm that focuses on minimizing application makespan using multiple rounds to improve overlap of communication with computation. The MI approach provides a closed-form solution for the scheduling problem, given a fixed number of rounds [13]. An important limitation is that a *linear cost model* is assumed, by which a chunk computation or communication is assumed to take a time exactly proportional to the chunk size (that is the number of independent load units in the chunk). In other words, the MI approach does not consider the latencies (i.e., start-up costs) that arise in all real-world platforms. In this paper, we focus on minimizing application makespan using multiple rounds, but we consider an *affine cost model* that incorporates both communication and computation latencies. Although this model has been used for single-round algorithms [14, 15, 16], it is an open question whether a closed-form multi-round schedule can be developed with affine costs. Furthermore, the work in [12] is only for homogeneous platforms, and it is also an open question to develop a multi-round algorithm that is applicable to heterogeneous platforms.

With linear cost models, the more rounds the lower the application makespan, as noted in [13]. By contrast, with affine cost models there is a clear trade-off: dividing the load into small chunks (i.e. many rounds) makes it possible to overlap communication with computation effectively, but dividing the load into large chunks (i.e. few rounds) reduces the overhead due to latencies, and thus the overall makespan. The implication is that there exists an *optimal number of rounds* for multi-round scheduling. Determining this optimum is again an open question.

In this paper we address the above three open questions for multi-round divisible load scheduling on platforms with a star topology. Our novel contributions are:

1. We obtain the first (to our knowledge) closed-form solution to the divisible load scheduling problem on homogeneous star platforms with affine costs, which are more realistic than the

previously used linear costs.

2. We propose a new multi-round scheduling algorithm, UMR (Uniform Multi-Round), which is applicable to heterogeneous platforms and computes a near-optimal number of rounds, with affine costs. We use simulation to compare UMR with previously proposed algorithms and demonstrate the benefits of our approach for wide ranges of scenarios.

This paper is organized as follows. In Section 2 we discuss relevant related work in detail. Section 3 describes our models for the application and the computing platform. Section 4 presents our closed-form solutions for multi-round divisible load scheduling on homogeneous platforms with affine cost models. Section 5 presents the UMR algorithm, which is then evaluated via simulation in Section 6. Finally, Section 7 concludes the paper and discusses future directions.

2 Related Work

Single-Round algorithms – Previous works on single-round algorithms defined recursion relations for chunk sizes that guarantee or approach an optimal schedule and that can be solved to obtain a closed-form solution to the scheduling problem on trees [17], linear arrays [18], 3-D mesh [19], or on hypercubes [20]. The asymptotic performance as the number of processors grows to infinity was studied on linear networks [21, 18], buses and trees [17], rings and 2-D meshes [22], and 3-D meshes [19]. While all the above works only consider homogeneous platforms, heterogeneous platforms are studied in [23, 12, 16].

The aforementioned works assume a linear cost model for communication and computation, but the work in [14] accounts for fixed latencies associated with network communication via an affine model, which is more realistic and has since been used in [24, 15]. The work in [15] also uses an affine cost model for computation. The introduction of affine costs renders the single-round scheduling problem significantly more complex on heterogeneous platforms. Some results are available when the platform is only partially heterogeneous [16]. For the general cases one must resort to Linear Programming [25].

Multi-Round algorithms – In spite of the known limitations of one-round algorithms, namely poor overlap of computation with communication, work on multi-round algorithms is rather limited. Proposed approaches belong in three categories: (i) those that focus on minimizing application makespan by improving overlap of communication with computation; (ii) those that focus on minimizing application makespan in the presence of performance prediction errors; and (iii) those that focus on maximizing steady-state application performance. Our work belongs to the first category, but we review all three categories below.

The first multi-round algorithm in category (i) is the Multi-Installment approach proposed in [13] for homogeneous platforms; little progress has been made in the area since then. Multi-Installment proceeds by dispatching chunks of work to compute resources in multiple rounds. The algorithm starts with small chunks and *increases* the chunk size throughout application execution to achieve effective overlap of communication and computation. In this paper we directly improve on the results in [13] for homogeneous platforms by considering latencies associated with computation and communication. Latencies naturally raise the question of the optimal number of rounds, which is not answered in [13]. We address this question both in the homogeneous and the heterogeneous case with a novel scheduling algorithm, UMR.

In the real world, the actual time a compute job or a transfer takes is always different than what we predict by some amount, either due to shared computing platforms or to non-deterministic applications. Multi-round algorithms that account for significant performance prediction errors were proposed in [26, 27]. Instead of increasing chunk size throughout application execution, these approaches start with large chunks and *decrease* chunk size throughout application execution. Chunks are dispatched to compute resources in a greedy fashion. The major disadvantage is that these algorithms can lead to very poor overlap of computation with communication. In [28] we have proposed an approach that combines UMR and Factoring [26], so it first increases and then decreases chunk size throughout application execution to achieve both effective overlap of computation with communication and robustness to performance prediction errors. In this paper we do not consider performance prediction errors.

Finally, multi-round algorithms have also been developed to maximize *steady-state* application performance [29, 30, 31]. They use identical rounds and the schedules are periodic. In this work we are solely concerned with minimizing application makespan.

3 Models

3.1 Application

Divisible load applications are characterized by input, or *load* that consists of large numbers of independent *units*, and can thus be divided into *chunks* that contain arbitrary numbers of units. The time for processing one unit is very small compared to that for processing the whole input, and it is assumed that the load, which we denote as W_{total} is arbitrarily and continuously divisible.

Many applications fall into the divisible load category. For example, bioinformatics applications, such as HMMER [32], take a query DNA/protein sequence and search it against a dictionary file containing millions of sequences, typically returning a few matching sequences. The dictionary file may be arbitrarily divided into many chunks and each sequence is a load unit. Volume rendering applications [9, 33, 7, 8, 6] also qualify as divisible loads. For example, the male specimen dataset from the Visible Human Project [9] contains 180MB (MR data), 730MB (CT data) and 62GB (photo data). These applications take N^3 amount of data and produce images of size N^2, and load units are voxels. MPEG video compression [34, 35] is also an example: input video is composed of large number of frames or Group of Pictures (GOP), each frame or GOP is a load unit that can be processed independently. While an input in raw DV format may take 13GB of space, the output is much smaller, ranging from 200MB$\sim$2GB depending on the compression rate. More examples include Radar data analysis [5], and Datamining [36, 37, 38].

To examine the spectrum of divisible load applications, we benchmarked three applications on

Table 1: Characteristics of divisible applications

Application	input size (MB)	running time (sec)	R
HMMER	802.0	534	6.7
MPEG	716.8	2494	34.8
VFleet	87.5	600	68.0
Data Mining	400.0	3150	78.0

a Athlon 1.8GHz machine: HMMER; Mencoder [39], a video compression tool; and Vfleet [40], a volume rendering software. We show in Table 1 the input size, running time, and the computation to communication ratio, R, if the input data were to be transferred over a 100Mb network. Data from a data mining application presented in [41] is also shown.

In this paper, we do not model transfer of output data back to the master. This is a common and perhaps surprising assumption made in previous work on multi-round divisible load scheduling [13]. (The work in [29] models output but only considers steady-state application performance as opposed to makespan minimization.) One rationale is that the output data size in many divisible load application is orders of magnitude smaller than input data size, as is the case with the applications we referenced earlier. The problem of deciding on an optimal way to return output to the master is open. In [42] we have provided a simple solution: return output from a round right before sending out input for the next round. This is straightforward to apply to multi-round algorithms including the ones presented in this paper, and performs reasonably well in practice.

3.2 Computing Platform

Our target computing platforms are clusters and grids that consist of multiple clusters. The input data is originally located on a single machine, the *master*. From the master's perspective the platform's logical topology is effectively a single-level tree/star (see Fig. 1). Let N be the number of *workers* in the platform.

The master sends out chunks to the workers over a network. We assume that the master uses its network connection in a sequential fashion: chunks are not sent to workers simultaneously. This is the common assumption in the literature and is justified by the behavior of the network (e.g. LAN), or local I/O bottlenecks. We discuss why one may consider removing this assumption in Section 7. We assume that workers can receive data from the network and perform computation simultaneously (conforming to the ”with front-end” model in [13]).

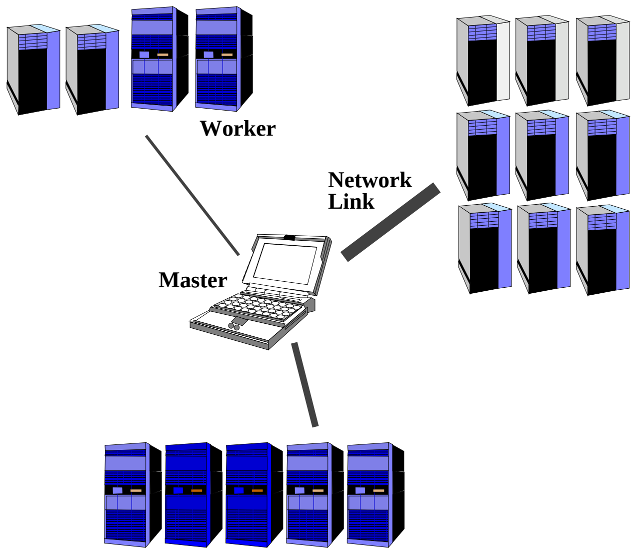

Figure 1: Computing platform model.

Consider a portion of the total load, $chunk_i \leq W_{total}$, which is to be processed on worker $i, 1 \leq i \leq N$. We model the time required for worker i to perform the computation, $Tcomp_i$, as

$$Tcomp_i = \alpha_i + \frac{chunk_i}{S_i}, \tag{1}$$

where α_i is a fixed latency, in seconds, for starting the computation, and S_i is the computational speed of the worker, in units of load per second. It is important to note that $chunk_i$ in the above equation is in units of load and not necessarily in bytes. Therefore, this equation does not imply that the computational complexity of a single unit of load has to be linear in the size of the unit in bytes. In fact, a unit may be one dictionary sequence for HMMER, one video frame for Mencoder, or one voxel for Volume rendering, and these the applications, described in Section 3.1, all exhibit various computational complexities. Because the units are independent, the execution time of a chunk is just the sum of the execution times of all its load units, and is proportional to the number of units, hence the $\frac{chunk_i}{S_i}$ term.

We model the time for sending $chunk_i$ units of load to worker i, $Tcomm_i$, as:

$$Tcomm_i = \beta_i + \frac{chunk_i}{B_i}, \tag{2}$$

where β_i is the latency, in seconds, incurred by the master to initiate a data transfer to worker i and B_i is the data transfer rate to worker i, in units of load per second.

Although previous work on multi-round divisible load scheduling [13] has used linear functions for $Tcomp$ and $Tcomm$ we use affine models as they are more realistic. The α component may be caused by the delay for starting a remote process; for instance, when establishing an ssh session or when accessing a resource via grid middleware services that involve resource acquisition, user authentication, service instantiation, etc. [43]. The β component includes the time to pre-process application data, to initiate a TCP connection or even an scp session, and to the physical network latency. Both components can be significant in practice. We have implemented a software environment to deploy divisible load applications on grid platforms [44]. We experienced values of α ranging from 0.1 to 0.7 seconds, and values of β ranging from 0.7 to 7 seconds, on a real-world grid testbed [44]. In fact, measurements reported by a recent grid benchmarking project [45] show values for α up to 45 seconds. We conclude that modeling both latencies is key to a realistic model.

Our platform model is flexible enough that it encompasses most previously used models in the divisible load literature that set both α and β to zero [13], only α to zero [14], or both α and β non-zero [15, 24]. To the best of our knowledge, only these two last works use a non-zero α, but only for a single-round algorithm.

4 Extension of Multi-Installment to Affine Costs

The MI algorithm in [13] only considers linear cost models. We evolve MI so that it can account for affine cost models for communication and computation and call the new algorithm XMI (eXtended

MI). In this section we consider a homogeneous star platform, meaning that $B_i = B$, $S_i = S$, $\alpha_i = \alpha$, and $\beta_i = \beta$ for all $1 \leq i \leq N$.

4.1 Chunk Size Recursion

We use T_{total} to denote the amount of time to process W_{total} units of load on a single worker. Let M be the total number of rounds, which should be given by the user as a parameter and not computed by XMI (this is precisely one disadvantage of MI/XMI that we address in Section 5.) Fig. 2 depicts the computation on 5 workers in 3 rounds, i.e. with 15 chunks of load. Chunk transfers from the master are shown in light gray boxes, whereas chunk computations are shown in white boxes, and latencies are shown in black and dark gray. For convenience, we number the chunks in the reverse order in which they are allocated to workers: the last chunk is numbered 0, the worker receiving the last chunk is numbered 0, and the last round is also numbered 0. Our goal is to compute the values of all chunk sizes, $chunk_i$, for $i = 0, \ldots, N \times M - 1$.

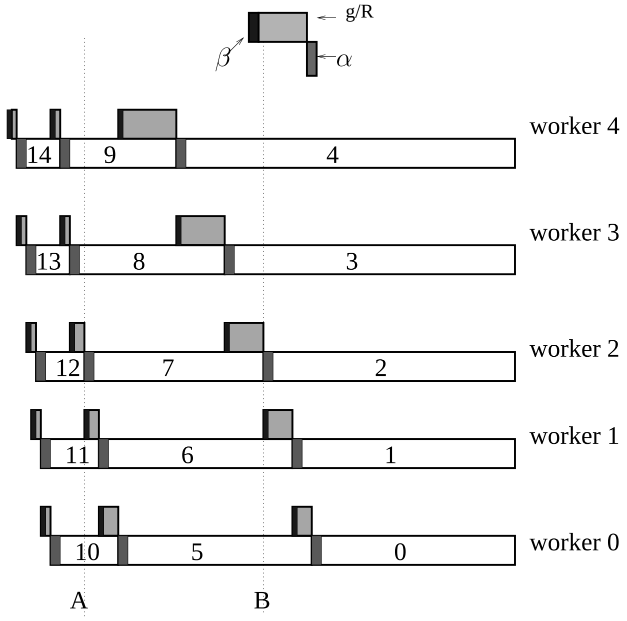

Figure 2: Illustration of the chunk size recursion.

Instead of developing a recursion on the $chunk_i$ series directly, we define $g_i = chunk_i/S$ as the time to compute the i^{th} chunk on a worker. Let us also define the computation-communication ratio of the platform, $R = B/S$. With these two definitions, $Tcomm_i$ and $Tcomp_i$ are:

$$Tcomm_i = \beta + chunk_i/B = \beta + g_i/R, \qquad \text{and} \qquad Tcomp_i = \alpha + g_i.$$

As in [13], so that both the network and the workers are kept as busy as possible, each worker must compute a chunk in exactly the time required for all the next N chunks to be sent to the workers:

$$\alpha + g_i = (g_{i-1} + g_{i-2} + \cdots + g_{i-N})/R + N \times \beta.$$

For example, in Fig. 2, we can write that while worker 2 computes chunk 7 (from time A to time B) chunks 6 to 2 must be sent to workers 1, 0, 4, 3, and 2: $\alpha + g_7 = (g_6 + g_5 + \cdots + g_2)/R + 5 \times \beta$. The above equation is only valid for $i \geq N$. For $i < N$ we need the following modification:

$$\alpha + g_i = (g_{i-1} + g_{i-2} + g_{i-3} + \cdots + g_{i-N})/R + i \times \beta + g_0 + \alpha,$$

where we let $g_i = 0$ for $i < 0$. We summarize our recursion as:

$$\forall\, i \geq N \qquad \alpha + g_i = (g_{i-1} + g_{i-2} + g_{i-3} + \cdots + g_{i-N})/R + N \times \beta, \tag{3}$$

$$\forall\, 0 \leq i < N \quad \alpha + g_i = (g_{i-1} + g_{i-2} + g_{i-3} + \cdots + g_{i-N})/R + i \times \beta + g_0 + \alpha, \tag{4}$$

$$\forall\, i < 0 \qquad g_i = 0. \tag{5}$$

Due to latencies, this recursion is more complex than the on in [13], but we will see that it is nevertheless amenable to an analytical solution.

4.2 Solving the Recursion

The recursion in the previous section can be solved via generating functions. We only present a sketch of the solution as it is only technical (the complete derivations are given in [42]). Let $G(x)$

be the generating function for the series g_i: $G(x) = \sum_{i=0}^{\infty} g_i x^i$. By multiplying Eq. 3 and Eq. 4 by x^i, summing for all $i \geq 0$, and using the well known property that the n^{th} coefficient of $\frac{G(x)}{1-x}$ is computed as $\sum_{i=0}^{n} g_i$, one obtains:

$$\begin{aligned} G(x) &= \frac{g_0(1-x^N) - \alpha \times x^N + \beta(x + x^2 + x^3 + ... + x^N)}{(1-x) - x(1-x^N)/R} \\ &= g_0 G'(x) + G''(x), \end{aligned}$$

where G' and G'' are two generating functions with the same denominator, $Q(x)$.

The simple rational expansion theorem [46] can be used to determine the coefficients of $G(x)$, given the roots of its denominator polynomial. $Q(x)$ has $N+1$ roots (one of these roots is 1). Let θ_j, $j = 0, ..., N$, be the inverses of these roots. The partial fraction and rational expansion theorem gives the g_i series as:

$$g_i = g_0 \sum_{j=0}^{N} \eta_j \theta_j^i + \sum_{j=0}^{N} \xi_j \theta_j^i, \tag{6}$$

where the η_j and the ξ_j series can be computed respectively for G' and for G'' as in [46]. One can compute g_0 by simply writing that the g_i series sums up to T_{total}. Note that the simple rational expansion theorem can only be applied if all roots are of degree 1. In [42] we proved that this is the case when $R \neq N$. When $R = N$, we proved that the only root of degree higher than 1 is the root $x = 1$, and it is of degree 2. In this specific case it is straightforward to apply the *general* rational expansion theorem, also given in [46]. In all cases, each chunk size, $g_i \times S$, turns out to be a linear combination of N geometric series. This completes our derivation of a closed-form solution for the XMI schedule on a homogeneous star platform with affine costs for both computation and communication. This is a direct improvement over the work in [13] that only considered linear costs. As explained in Section 1, the introduction of affine costs raises a critical question: what is the optimal number of rounds? Next, we introduce a novel scheduling algorithm that not only determines a near-optimal number of rounds but is also applicable to heterogeneous platforms.

5 The UMR Algorithm

In this section we propose a novel multi-round scheduling algorithm: Uniform Multi-Round (UMR). Like XMI, UMR increases the chunk size in between rounds to reduce the overhead due to communication and computation latencies. However, UMR imposes the restriction that rounds be "uniform", meaning that chunk sizes are *fixed* within each round. This restriction, while precluding optimal overlap of communication with computation, makes it possible to compute a near-optimal number of rounds, which we denote by M^*. We found that the algebraic solution of XMI is too complex for computing M^*. Instead, with the uniform round restriction we can compute it and obtain a schedule that is reasonably close to the XMI-M^* schedule. This intuition also comes partly from the uniform round concept introduced in [26] in which the Factoring algorithm sends out chunks in uniform rounds of *decreasing* sizes to dynamically allocate to workers. Factoring was designed without considering communication delays but considering uncertainty on chunk computation time. In this work we consider communication delay but assume no uncertainty on computation times. This led us to still using uniform round with increasing chunk sizes. Finally, and perhaps most important, UMR is applicable to heterogeneous platforms, unlike previously proposed multi-round algorithms. In this section, M denotes the number of rounds to be computed by UMR. We first describe the UMR algorithm for homogeneous platforms.

5.1 UMR on Homogeneous Platforms

Induction on chunk sizes – Let $chunk_j$, for $j = 0, .., M-1$, be the chunk size used for all workers at round j. We illustrate the operation of UMR in Fig. 3. At time T_A, the master starts dispatching chunks of size $chunk_{j+1}$ for round $j+1$ while the workers are performing computations for round j. Platform utilization is maximized if the time for worker N to compute the round j chunk is equal to the time for the master to send work for round $(j+1)$ to all N workers (from

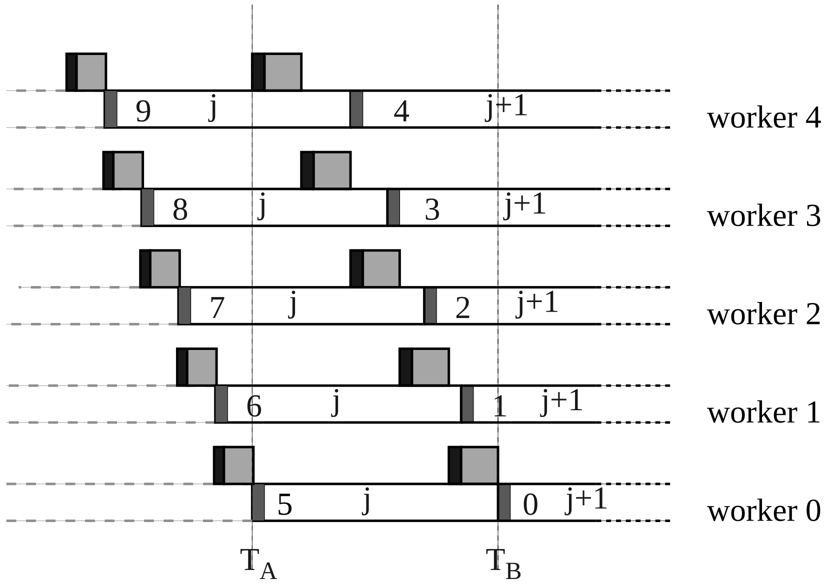

Figure 3: UMR dispatches the load in rounds, where the chunk size is fixed within a round and increases between rounds.

time T_A to time T_B in Fig. 3). Therefore, one can write:

$$\alpha + \frac{chunk_j}{S} = N(\frac{chunk_{j+1}}{B} + \beta). \tag{7}$$

The left-hand side of Eq. 7 is the time worker N spends initiating and performing a chunk computation during round j. The right-hand side is the time it takes for the master to send data to all N workers during round $j+1$. One can then distinguish two cases for deriving $chunk_j$ given the simple induction defined in Eq. 7:

$$\begin{cases} \text{if} \quad NS \neq B \quad \text{then} \quad \forall j \;\; chunk_j = (\frac{B}{NS})^j(chunk_0 - \Delta) + \Delta, \;\; \text{where } \Delta = \frac{BS}{B-NS}(N \times \beta - \alpha). \\ \text{if} \quad NS = B \quad \text{then} \quad \forall j \;\; chunk_j = chunk_0 + jS(\alpha - N\beta). \end{cases} \tag{8}$$

We have thus obtained a geometric series of chunk sizes when $NS \neq B$, and an arithmetic series when $NS = B$, where $chunk_0$ is an unknown. If we compare Eq. 8 with Eq. 3, we can see that UMR essentially satisfies Eq. 3 only for the last chunk in a round. For example, in Fig 3, the compute time of chunk 5 is equal to the transfer time of chunk 0 $\sim$ chunk 4, and satisfy Eq 3, but the compute time of chunk 6 is greater than the transfer time of chunks 1 to 5.

Constrained minimization problem – The objective of our algorithm is to minimize $Ex(M, chunk_0)$, the makespan of the application:

$$Ex(M, chunk_0) = \frac{W_{total}}{NS} + M \times \alpha + \frac{1}{2} \times N(\beta + \frac{chunk_0}{B}).$$

The first term is the time for worker N to perform its computation. The second term is the overhead incurred at each round to initiate a computation. The third term corresponds to the time for the master to send all the data for round 0. The $\frac{1}{2}$ factor is due to an optimization for the last round, during which the master allocates chunks of decreasing sizes to the workers to ensure that they all finish computing at the same time. This is exactly the same approach as used for the last round of the MI and XMI algorithms and details are provided in a technical report [47].

We also have the constraint that the amount of work sent out by the master during the execution sums up to the entire load:

$$G(M, chunk_0) = \sum_{j=0}^{M-1} N \times chunk_j - W_{total} = 0.$$

This constrained minimization problem, with M and $chunk_0$ as unknowns, can be solved with the Lagrange Multiplier method [48]. The multiplier, $L(chunk_0, M, \lambda)$, is defined as:

$$L(chunk_0, M, \lambda) = Ex(M, chunk_0) + \lambda \times G(M, chunk_0),$$

and we must solve:

$$\begin{cases} \frac{\partial L}{\partial \lambda} = G = 0 \\ \frac{\partial L}{\partial M} = \frac{\partial Ex}{\partial M} + \lambda \times \frac{\partial G}{\partial M} = 0 \\ \frac{\partial L}{\partial chunk_0} = \frac{\partial Ex}{\partial chunk_0} + \lambda \times \frac{\partial G}{\partial chunk_0} = 0. \end{cases}$$

This system of equations reduces to the following equations for M:

$$\begin{cases} \text{if } \ NS \neq B \quad \text{then} \quad N\Delta - \frac{W_{total} - NM\Delta}{1-(\frac{B}{NS})^M}\left(\frac{B}{NS}\right)^M \ln\left(\frac{B}{NS}\right) - 2\alpha \times B\frac{1-(\frac{B}{NS})^M}{1-\frac{B}{NS}} = 0, \\ \text{if } \ NS = B \quad \text{then} \quad M = \sqrt{(\frac{2NW_{total}}{B(3\alpha+N\beta)})}. \end{cases} \tag{9}$$

The first equation can be solved numerically by bisection. The solve is fast (on the order of 0.017 seconds on a 1.6GHz Athlon) and can thus be implemented in a runtime scheduler with negligible overhead. Once we have computed M^*, the solution to Eq. 9, $chunk_0$ follows as:

$$\begin{cases} \text{if } \ NS \neq B \quad \text{then} \quad chunk_0 = \frac{(1-\frac{B}{NS})(W_{total}-NM^*\Delta)}{N\times(1-(\frac{B}{NS})^{M^*})} + \Delta, \\ \text{if } \ NS = B \quad \text{then} \quad chunk_0 = S(2M^*\alpha - (M^* - \frac{1}{2})(\alpha - N\beta)). \end{cases} \tag{10}$$

Finally, the $chunk_j$ series can now be computed with Eq. 8. Complete details on these derivations are provided in a technical report [47]. In that technical report we also develop necessary conditions for all workers to be utilized: the smaller the computation-computation ratio, the fewer the number of workers that can be utilized effectively. Simply put, it is beneficial to use N workers only when $N \leq R$, where R is the computation-communication ratio B/S. Otherwise, just reduce N. In all our experimental results we ensure that all workers are utilized.

5.2 UMR on Heterogeneous Platforms

The analysis of UMR in the heterogeneous case is more involved than that for the homogeneous case but follows exactly the same steps. While in the homogeneous case we fixed the chunk size at a round, in the heterogeneous case we fix the *time* it takes for each worker to compute a chunk during a round: $\alpha_i + chunk_{ji}/S_i = t_j$ for all $i = 1, .., N$ where $chunk_{ji}$ is the amount of load sent out to worker i during round j, and quantity t_j depends only on j. Let $round_j = \sum_{i=1}^{n} chunk_{ji}$ be the amount of load processed during round j. By these two definition it is straightforward to express $chunk_{ji}$ as an affine function of $round_j$. One can then write an equation Similar to Eq. 7

to express the fact that the master sends round $j+1$ data to all N workers while worker N performs its round j computation:

$$\sum_{i=1}^{N}(\frac{chunk_{j+1,i}}{B_i} + \beta_i) = t_j.$$

After replacing $chunk_{j+1,i}$ and t_j in the above equation by their expressions in terms of $round_j$, one obtains:

$$round_j = \theta^j \times (round_0 - \eta) + \eta,$$

where η and θ are constants depending on platform parameters $\alpha_i, \beta_i, S_i, B_i \quad (1 \leq i \leq N)$. As for the homogeneous case, we have a constrained optimization problem:

$$\begin{array}{ll} \text{minimize} & Ex(M, round_0) = \sum_{j=0}^{M-1} t_j + \frac{1}{2} \sum_{i=1}^{N} \left(\frac{chunk_{0,i}}{B_i} + \beta_i \right), \\ \text{subject to} & G(M, round_0) = M\eta + \frac{round_0 - \eta}{1-\theta} \times (1 - \theta^M) - W_{total} = 0, \end{array}$$

which can be solved with the Lagrange multiplier method. Since the derivation is purely technical but rather cumbersome, all details are provided in a technical report [42].

On an heterogeneous platform, resource selection is needed when the full platform cannot be utilized effectively (in the homogeneous case one can just reduce the value of N). For the heterogeneous case UMR implements a simple resource selection criterion, which is inspired by the work in [30]: processors with faster network connections are selected first. We evaluate how UMR perform on heterogeneous platforms in Section 6.3.

6 Evaluation of UMR

To evaluate UMR we developed a simulator with the SIMGRID [49] toolkit, which provides the necessary tools and abstractions for studying scheduling strategies for parallel applications on distributed platforms. We used our simulator for three sets of experiments. First, we compared UMR to the XMI algorithm developed in Section 4.1. Second, we study the impact of system parameters (S, B, α, and β) on UMR's choice for the optimal number of rounds. Third, we

Table 2: Parameter values for the experiments presented in Section 6.1.

Parameter	Values
Number of processors	$N = 5, 10, 15, \ldots, 50$
Workload (unit)	$W_{total} = 2000$
Compute rate (unit/s)	$S = 1$
Comp./comm. ratio	$R = N, N+2, N+4, \ldots, 80$
Computation latency (s)	$\alpha = 0.0, 0.5, \ldots, 10$
Communication latency (s)	$\beta = 0.0, 0.5, \ldots, 10$

evaluate how robust UMR is to platform heterogeneity.

6.1 Comparison with Previous Algorithms

In [50] we had compared an early version of UMR with the MI approach [13] in scenarios with both computation and communication latencies. This comparison was in some sense unfair as MI ignores latencies while UMR takes them explicitly into account. Now that we have developed XMI, which is strictly superior to MI, we revisit our earlier comparison. The goal of our simulations is to answer the following question: does UMR's ability to compute a near-optimal number of rounds outweigh the penalty due to the restriction it imposes on chunk sizes, when compared with the XMI algorithm? Note that, to the best of our knowledge, UMR is the first multi-round algorithm to support heterogeneous platforms, and is in this sense inherently superior to XMI. Nevertheless, we present simulation results on homogeneous platforms to be able to answer the question above.

Since XMI does not compute a number of rounds, we use XMI with 1 to 8 rounds, denoted as XMI-x for $x = 1, \ldots, 8$. Note that XMI-1 is identical to the one-round algorithm proposed in [24].

6.1.1 Experimental Scenario

We evaluate UMR and XMI-x for the range of parameter values in Table 2, which correspond to the spectrum of real-world application data shown in Table 1 and to the range of α and β values observed in practice (see Section 3.2). The compute speed is set to 1 load unit/second, and the total load is set to 2000 units, so that the total execution time corresponds to 2000 seconds, roughly

Table 3: Comparison between UMR and XMI-x, averaged over 9,529,110 experiments.

	UMR	XMI-1	XMI-2	XMI-3	XMI-4	XMI-5	XMI-6	XMI-7	XMI-8
Normalized Makespan	1.00	1.03	1.10	1.49	1.68	1.82	1.94	2.06	2.16
% Degradation from Best	0.88	2.85	9.37	40.43	59.11	74.09	86.90	99.21	110.00

falling in the middle of the range of execution times in Table 1. We vary N and R to explore a range of scenarios with $N <= R$, ensuring that all workers can be utilized (see the discussion at the end of Section 5.1). For each instantiation of these parameters we simulated UMR and XMI-x, and computed the makespans they achieved.

6.1.2 Aggregate Results

Table 3 shows the comparison between UMR and XMI-x, averaged over all parameter configurations. The first row shows the ratio of makespan achieved by XMI-x to that achieved by UMR; the second row shows the percentage degradation from best. This metric is commonly used in the scheduling literature: at each parameter instantiation, compute for each algorithm how far the makespan achieved by that algorithm is from the best algorithm considered for that parameter instantiation, in percentage; take the average over all parameter instantiations.

The main observation is that UMR outperforms XMI-x on average: all XMI-x have an average makespan higher than that of UMR, and UMR has the lowest average degradation from best (by $\approx$ 2%). Over all the experiments, UMR is the best algorithm in 66.57% of the cases. In the cases where UMR is not the best, it is on average within 2.64% of its competitors, with a standard deviation of 4.52%. On average the best XMI-x algorithm is XMI-1, but we will see that this is not true in all regions of our parameter space.

6.1.3 Impact of Computation/Communication Ratio on Makespan

For each value of R we compute the makespan of XMI-x normalized to that achieved by UMR, averaged over all other parameters. We only discuss XMI-x results for $x = 1, \ldots, 4$ as trends are

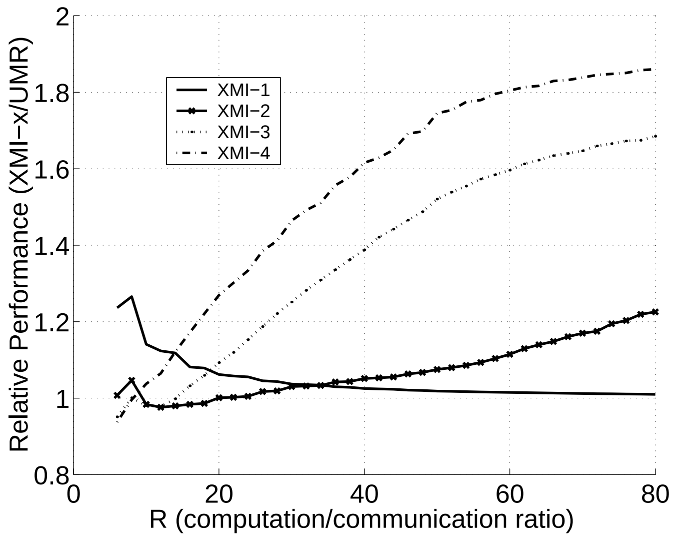

Figure 4: XMI-x makespan normalized to UMR makespan vs. R.

identical for $x \geq 4$. Results are shown in Fig. 4.

Results for $\alpha \geq 0$ and $\beta \geq 0$ – At lower values of R, communication is relatively expensive, and it is wise to use more rounds to start up quickly, as is shown in the figure; at $R \approx 10$, XMI-$3, 4$ are better than XMI-$1, 2$. They are even better than UMR as they do not have the uniform round restriction and thus overlap communication and computation better. But as R increases, communication becomes less critical, and latencies play and increasingly important part in the overhead of communication. As a result, XMI-$3, 4$ degrade when R increases, while the makespan of XMI-1 gradually drops, and that of XMI-2 initially drops and then increases after $R > 12$. Over the entire range of R, we see that UMR achieves the lowest makespan or one close to the lowest.

Results for $\alpha = 0$ and $\beta = 0$ – We just mentioned that XMI overlaps communication and computation better than UMR. To give a perspective of how much different they are in this respect, we compared their makespans when there are no latencies. Under this condition the optimal number of rounds is ∞. To enable a fair comparison we limit XMI-x to $x \leq 8$ in our experiments and force UMR to use the same number of rounds x as XMI-x. We found that XMI-x indeed outperforms UMR because it is not restricted to using uniform rounds and can therefore achieve better

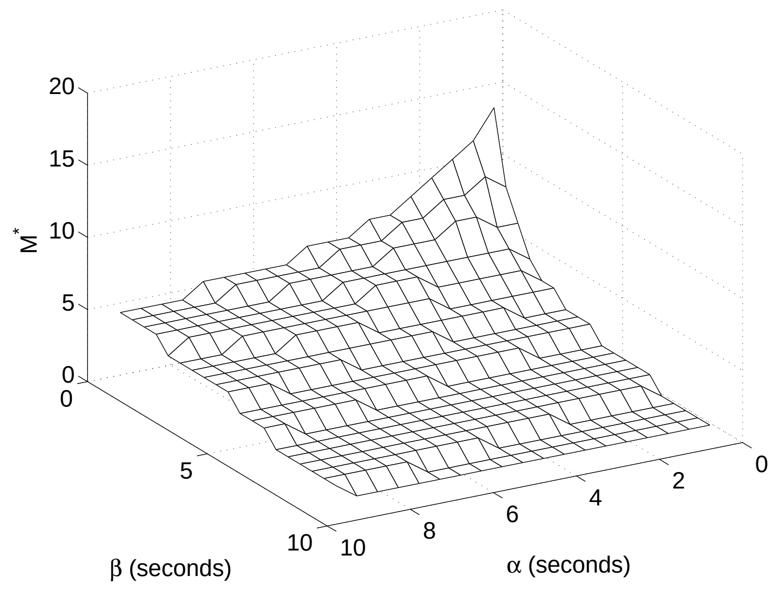

Figure 5: M^* versus β and α.

overlap between computation and communication. However, UMR is only within 1.6% of XMI-x on average. This quantifies the net performance penalty due to using uniform rounds.

6.2 Impact of System Parameters on M^*

UMR is able to improve over previously proposed algorithms in spite of the uniform round restriction, and precisely because this restriction makes it possible to compute an optimal number of rounds. In this section we study how the optimal number of rounds varies with system parameters.

6.2.1 Impact of Latencies on M^*

Fig. 5 plots the M^* value chosen by UMR versus both α and β when they vary between 0 and 10. The other parameters are fixed and set to $N = 5$, $W_{total} = 2000$, and $R = 5$. These parameter values correspond to cases in which it may be beneficial to use multiple rounds. For smaller R values, for instance, UMR may always use $M^* = 1$. Fig. 5 demonstrates that UMR chooses different values of M^* for different scenarios, in this case between 4 and 10 rounds. M^* decreases when either the communication or computation latency increases, which is expected as fewer rounds lead to less overhead.

One may wonder why UMR reduces M^* at all when β increases since the communication latency is hidden after the first round. As β increases, larger chunks need to be sent during the first round so that the first worker finishes computing after the communication to the last worker has been completed. Therefore, $chunk_0$ has to be larger and since the series of chunk sizes is increasing, fewer rounds are necessary to dispatch the entire load. Hence the decrease in M^*.

6.2.2 Impact of the Computation/Communication Ratio on M^*

Fig. 6 plots M^* versus R ratio for three different (α, β) values. Two different mechanisms are at play here:

#1. $\underline{\alpha = 5, \beta = 0}$: When β is insignificant, the only overhead is the one incurred by computation at each round, which is minimized by using as few rounds as possible. But when R is low, that is when communication is relatively expensive, it is beneficial to use more rounds for better overlapping of communication with computation. This is seen on the black solid line in Fig. 6, with M^**decreasing* from 6 to 2 as R increases.

#2. $\underline{\alpha = 0, \beta = 5}$: When α is insignificant, the only overhead is the one due to worker idle time while waiting for round 0 data, which is minimized by a small $chunk_0$ value and a large number of rounds. However, UMR must send out sufficient load to workers in the first round so that they are kept busy computing while data transfers take place. More specifically, we see in Eq. 8 that $chunk_0$ must be greater than $\Delta = R(N\beta - \alpha)/(R - N)$, which decreases with R and increases with β. Therefore, when β is significant, UMR is forced to use a relatively large value for $chunk_0$ at low R, which prevents the use of many rounds. More rounds can be used as R increases. This is seen on the black dashed line in Fig. 6, with M^**increasing* from 2 to 14 as R increases.

The grey line in Fig 6 ($\alpha = \beta = 5$) shows a combination of these two effects with first an increase and then a decrease of the number of rounds. At low R, the overhead due to the network

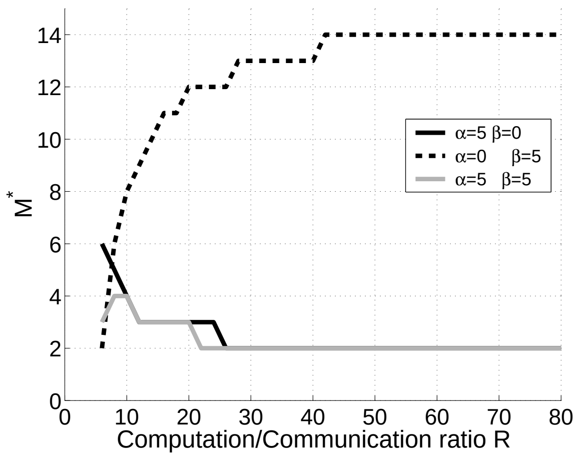

Figure 6: Effect of the computation/communication ratio on M^*. $W_{total} = 2000$, $N = 5$.

communications in round 0 is dominant, and UMR increases the number of round with R as in item #2 above. In this experiment, when $R \approx 10$ the overhead due to computation latencies is now dominant, and UMR starts using fewer and fewer rounds to limit its impact as in item #1 above.

6.3 UMR on Heterogeneous Platform

To show that UMR works well on heterogeneous platforms, we simulated UMR on a platform consisting of 10 processors with $W_{total} = 2000$ and random S_i, α_i, β_i and B_i values sampled from a uniform distribution on the interval $((1 - \frac{het-1}{1+het})mean, (1 - \frac{het-1}{1+het})mean$, where $mean$ is 1, 1, 1, and 20 for S_i, α_i, β_i, and B_i, respectively. In other words, processor and link characteristics can differ by as much as a factor of het between processors.

Fig. 7 plots the normalized makespan achieved by UMR versus het (solid curve). The normalized makespan is computed as the ratio of the makespan versus the "ideal" makespan which could only be achieved if all communication costs were zero, that is $W_{total}/\sum S_i$. Every data point in the figure is obtained as an average over 100 samples. The solid line show results with the resource selection scheme described in Section 5.2. One can see that UMR is robust and handles heterogeneous platforms well. For extreme cases in which processor or link performances differ by a factor up to 1,000, UMR still managed to achieve a makespan which is within 30% of the ideal. For comparison, the dotted lines shows the normalized makespan when no resource selection scheme

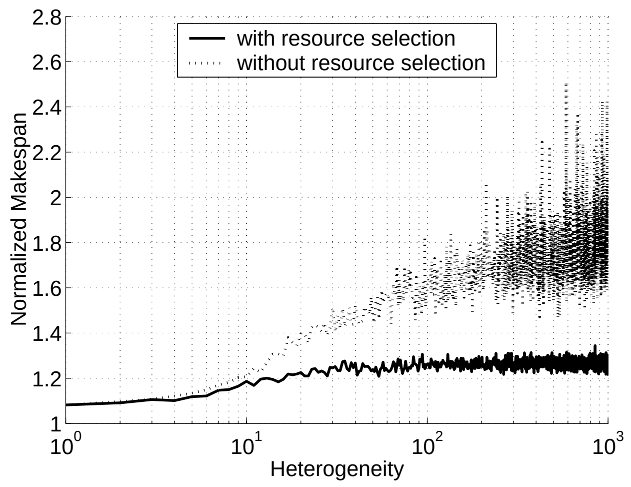

Figure 7: Heterogeneous platform, normalized makespan versus het, with and without resource selection.

is used, showing that our resource selection scheme is effective.

7 Conclusion

In this paper we have presented multi-round algorithms for scheduling divisible loads on star networks. We set out to answer three open questions: (i) given a fixed number of rounds, is it possible to obtain a closed-form optimal schedule analogous to the ones given in [13] for homogeneous platforms, but with computation and communication latencies? (ii) is it possible to design an effective multi-round algorithm applicable to heterogeneous platforms; and (ii) is it possible to compute an optimal number of rounds? Our contributions are as follows. First, building on the work in [13], we have developed the XMI algorithm, which provides a new closed-form solution for multi-installment scheduling on homogeneous star platforms with affine cost models, i.e. with communication and computation latencies. Second, we have introduced a new multi-round algorithm, UMR (Uniform Multi-Round), which sends a fixed amount of work to each worker within each round. This restriction makes it possible to compute a near-optimal number of rounds, which was not possible for previously proposed algorithms. Furthermore, to the best of our knowledge, UMR is the first proposed multi-round algorithm that is amenable to heterogeneous platforms. Our simulation results demonstrate that UMR's ability to compute a near-optimal number of rounds outweighs on average the penalty due the restriction it imposes on chunk sizes, when compared with the XMI algorithm. We also showed that UMR utilizes heterogeneous platforms effectively.

The above contributions make it possible to achieve our goal of implementing multi-round divisible load scheduling algorithms in practice. In this paper we have assumed that the times required for chunk transfers and computations are perfectly predictable. This common assumption often breaks down in real-world situations and we have revisited it in our most recent work to design a more robust version of UMR [28]. We have implemented this robust UMR as part of a grid application execution environment [44] that leverages our research on scheduling algorithms to deploy divisible load applications in practice. One future direction is to allow the master to perform simultaneous communications to workers. This can be beneficial on wide area networks due to bandwidth-sharing properties [51] that make it possible to achieve higher throughput with parallel TCP streams. This will require the development of new multi-round scheduling algorithms.

References

[1] Jr. E.G. Coffman, M.R. Garey, and D.S. Johnson. *Bin Packing Approximation Algorithms: A Survey*. PWS Publishing Co., Boston, MA., 1996.

[2] M. Drozdowski and P. Wolniewicz. Experiments With Scheduling Divisible Tasks in Clusters of Workstations. In *International Conference on Paralle and Distributed Computing (Europar)*, pages 311–319, 2000.

[3] V. Bharadwaj and S. Ranganath. Theoretical and Experimental Study on Large Size Image Processing Applications Using Divisible Load Paradigm on Distributed Bus Networks. *Image and Vision Computing*, 20(13-14):917–1034, 2002.

[4] C.K. Lee and M. Hamdia. Parallel Image Processing Applications on a Network of Workstation. *Parallel Computing*, 21:137–160, 1995.

[5] Graig Miller, David G. Payne, Thanh N. Phung, Herb Siegel, and Roy Williams. Parallel Processing of Spaceborne Imaging Radar Data. In *Proceedings from The International Conference for High Performance Computing and Communications (SC'95)*, 1995.

[6] Yi-Jen Chiang, Ricardo Farias, Claudio T. Silva, and Bin Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proceedings of the IEEE Symposium on Parallel and Large-data Visualization and Graphics*, pages 59–66, 2001.

[7] Wes Bethel, Brian Tierney, Jason lee, Dan Gunter, and Stephen Lau. Using High-speed WANs and Network Data Caches to Enable Remote and Distributed Visualization. In *Proceedings of The International Conference for High Performance Computing and Communications (SC'00)*, 2000.

[8] Antonio Garcia and Hen-Wei Shen. Parallel volume rendering: An interleaved parallel volume renderer with PC-clusters. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 51–59, 2002.

[9] Visible human project. `http://www.nlm.nih.gov/research/visible/visible_human.html`.

[10] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996.

[11] Special issue on *divisible load scheduling*. Cluster Computing, 6, 1, 2003.

[12] V. Bharadwaj, D. Ghose, and V. Mani. Optimal Sequencing and Arrangement in Single-Level Tree Networks With Communication Delays. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):968–976, 1994.

[13] V. Bharadwaj, D. Ghose, and V. Mani. Multi-Installment Load Distribution in Tree Networks With Delays,. *IEEE Trans. on Aerospace and Electronc Systems*, 31(2):555–567, 1995.

[14] J. Blazewicz and M. Drozdowski. Distributed Processing of Divisible Jobs With Communication Startup Costs. *Discrete Applied Mathematics*, 76:21–41, 1997.

[15] V. Bharadwaj, X. Li, and C.C. Ko. on The Influence of Start-Up Costs in Scheduling Divisible Loads on Bus Networks. *IEEE Transactions on Aerospace and electronic systems*, 11(12):1288–1305, 2000.

[16] O. Beaumont, A. Legrand, and Y. Robert. Optimal Algorithms for Scheduling Divisible Workloads on Heterogeneous Systems. Technical Report 2002-36, Ecole Normale Superieure de Lyon, October 2002.

[17] S Bataineh, T.-Y. Hsiung, and T.G. Robertazzi. Closed Form Solutions for Bus and Tree Networks of Processors Load Sharing a Divisible Job. *IEEE Transations on Computers*, 43(10), 1994.

[18] K. Li. Scheduling Divisible Tasks on Heterogeneous Linear Arrays With Applications to Layered Networks. In *Proceedings of the 3rd International Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications*, 2002.

[19] M. Drozdowski and W. Glazek. Scheduling Divisible Loads in a Three-Dimensional Mesh of Processors. *Parallel Computing*, 25(4), 1999.

[20] K. Li. Parallel Processing of Divisible Loads on Partitionable Static Interconnection Networks. *Cluster Computing*, 6(1):47–55, 2003.

[21] D. Ghose and V. Mani. Distributed Computationa With Communication Delays: Asymptotic Performance Analysis. *Journal of Parallel and Distributed Computing*, 23(3), 1994.

[22] J. Blazewicz. Performance Limits of a Two-Dimensional Network of Load Sharing Processors. *Foundations of Computing and Decision Sciences*, 21(1):3–15, 1996.

[23] H.-J. Kim, G.-I. Jee, and J.-G. Lee. Optimal Load Distribution for Tree Network Processors. *IEEE Transactions on Aerospace and Electronic Systems*, 32(2):607–611, 1996.

[24] A.L. Rosenberg. Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better. In *Proceedings from 3rd IEEE International Conference on Cluster Computing (CLUSTER'01)* , pages 124–131, 2001.

[25] M. Drozdowski and P. Wolniewicz. Divisible Load Scheduling in Systems With Limited Memory. *Cluster Computing*, 6(1):19–29, 2003.

[26] S.F. Hummel. Factoring : a Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(8):90–101, August 1992.

[27] T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journal of Parallel and Distributed Computing*, 47:185–197, 1997.

[28] Y. Yang and H. Casanova. RUMR: Robust Scheduling for Divisible Workloads. In *Proceedings of the 12th IEEE Symposium on High-Performance Distributed Computing (HPDC-12)*, pages 114–125, June 2003.

[29] D. Altilar and Y. Paker. An Optimal Scheduling Algorithm for Parallel Video Processing. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 245–248, 1998.

[30] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, June 2002.

[31] O. Beaumont, A. Legrand, and Y. Robert. The Master-Slave Paradigm With Heterogeneous Processors. Technical Report RR2001-13, Ecole Normale Superieure de Lyon, March 2001.

[32] HMMER Webpage. `http://hmmer.wustl.edu/hmmer-html/`.

[33] Alan Watt. *3D Computer Graphics*, chapter 13. Addison-Wesley, 3 edition, 2000.

[34] Ke Shen, Lawrence A. Rowe, and Edward J. Delp. A Parallel Implementation of an MPEG1 Encoder: Faster than Real-Time! In *Proceedings of the SPIE Conference on Digital Video Compression: Algorithms and Technologies*, pages 407–418, February 1995.

[35] F. J. Gonzalez-Castãno and R. Asorey-Cacheda and R. P. Martinez-Alvarez and F. Comesaña-Seijo and J. Vales-Alonso. DVD Transcoding via Linux Metacomputing. *Linux Journal*, 116:8, 2003.

[36] Nuno Amano, Joao Gama, and Fernando Silva. Exploiting Parallelism in Decision Tree Induction. In *Proceedings from the ECML/PKDD Workshop on Parallel and Distributed computing for Machine Learning*, pages 13–22, September 2003.

[37] Sanjay Goil and Alok Choudhary. High performance multidimensional analysis of large datasets. In *Proceedings of the 1st ACM international workshop on Data warehousing and OLAP*, pages 34–39, 1998.

[38] David Skillicorn. Strategies for Parallel Data Mining. *IEEE Concurrency*, 7(4):26–35, 1999.

[39] Mencoder media player. `http://www.mplayerhq.hu`.

[40] VFleet Webpage. `http://www.psc.edu/Packages/VFleet_Home`.

[41] M. Tamura, T. an Oguchi and M. Kitsuregawa. Parallel database processing on a 100 Node PC cluster: cases for decision support query processing and data mining. In *Proceedings from The International Conference for High Performance Computing and Communications* , pages 1–16, November 1997.

[42] Y. Yang and H. Casanova. Extensions to The Multi-Installment Algorithm: Affine Costs and Output Data Transfers. Technical Report CS2003-0754, Dept. of Computer Science and Engineering, University of California, San Diego, July 2003.

[43] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer*, 35(6), 2002.

[44] Krjn van der Raadt and Yang Yang and Henri Casanova. APST-DV: Divisible Load Scheduling and Deployment on the Grid. Technical Report CS2004-0785, Dept. of Computer Science and Engineering, University of California, San Diego, April 2004.

[45] Chun, G. and Dail, H. and Casanova, H. and Snavely, A. Benchmark Probes for Grid Assessment. In *Proceedings of the High-Performance Grid Computing Workshop*, April 2004.

[46] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics*. Wiley, 1994.

[47] Y. Yang and H. Casanova. Multi-Round Algorithm for Scheduling Divisible Workload Applications: Analysis and Experimental Evaluation. Technical Report CS2002-0721, Dept. of Computer Science and Engineering, University of California, San Diego, 2002.

[48] D. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, Belmont, Mass., 1996.

[49] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: The SIMGRID Simulation Framework. In *Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, May 2003.

[50] Y. Yang and H. Casanova. UMR: a Multi-Round Algorithm for Scheduling Divisible Workloads. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003.

[51] H. Casanova. Modeling Large-Scale Platforms for the Analysis and the Simulation of Scheduling Strategies. In *Proceedings of the 6th Workshop on Advances in Parallel and Distributed Computational Models*, April 2004.