RUMR: Robust Scheduling for Divisible Workloads

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

[1] *Department of Computer Science and Engineering*
[2] *San Diego Supercomputer Center*
University of California at San Diego

[yangyang,casanova]@cs.ucsd.edu

Abstract

Divisible workload applications arise in many fields of science and engineering. They can be parallelized in master-worker fashion and relevant scheduling strategies have been proposed to reduce application makespan. Our goal is to develop a practical *divisible workload scheduling strategy. This requires that previous work be revisited as several usual assumptions about the computing platform do not hold in practice. We have partially addressed this concern in a previous paper via an algorithm that achieves high performance with realistic resource latency models. In this paper we extend our approach to account for performance prediction errors, which are expected for most real-world performance and applications. In essence, we combine ideas from multi-round divisible workload scheduling, for performance, and from factoring-based scheduling, for robustness. We present simulation results to quantify the benefits of our approach compared to our original algorithm and to other previously proposed algorithms.*

1 Introduction

Applications that consist of many independent computational tasks arise in many domains [1, 2, 3, 4] and are well suited to master-worker execution on cluster platforms. In this paper we address the problem of scheduling these applications with the goal of reducing execution time, or *makespan*. This problem has been studied for two different scenarios: *fixed-sized tasks* and *divisible workload*. In the former scenario, the application's workload consists of tasks whose size (i.e. amount of required computation) are pre-determined, and a number of efficient scheduling strategies have been proposed [5, 6, 7, 8]. In this work we focus on the latter scenario, in which the scheduler can partition the workload in arbitrary, continuous "chunks" (in practical situations, this often means that the application consists of many similar computational tasks).

Examples of such applications are : feature extraction, in which a big image is segmented , and each segment is transfered to a worker and processed locally; Signal processing, which tries to recover a signal buried in a large file recording measurements; and sequence matching, [2], in which a single sequence is compared to a big dictionary file, and the running time is proportional to the letters in that dictionary.

The main question in scheduling divisible workload is how to choose an optimal division of the workload into chunks. One possible approach is to divide the workload in as many chunks as processors and to dispatch work in a *single round* of allocation [9, 10, 11, 3, 12]. This has several drawbacks, namely poor overlap of communication and computation and poor robustness to performance prediction errors. Consequently, a number of researchers have investigated *multi-round* algorithms. Three main observations have been made: (i) dividing the workload into large chunks reduces overhead, and thereby application makespan; (ii) the use of small chunks at the onset of application execution makes it possible to overlap overhead with useful work more efficiently; and (iii) the use of small chunks at the end of the execution leads to better robustness to performance prediction errors. Based on observations (i) and (ii), multi-round algorithms that use *increasing* chunk sizes throughout application execution to achieve good computation communication overlap while not suffering from prohibitive overheads [13]. Based on observations (i) and (iii), algorithms that use *decreasing* chunk sizes have been designed to tolerate performance prediction errors [14, 15]. The main contribution of this paper is to combine both approaches.

Our ultimate goal is to develop a scheduling strategy for divisible workloads that can be use in practice, i.e. as part of an application execution environment [16] run-

ning on real-world platforms. Our first step was to revisit multi-round scheduling algorithms that use increasing chunk sizes but with more realistic assumptions about the platform than in previous work, while still assuming that performance predictions are perfectly accurate. In [17, 13] we presented a novel multi-round algorithm, UMR (Uniform Multi-Round), that functions with realistic compute and network latency models, and that outperforms previously proposed approaches. This paper takes the next step and addresses the critical issue of performance prediction errors that arise due to uncertainties about the platform and the application. We present an extension to UMR: RUMR (Robust Uniform Multi-Round). RUMR borrows from UMR and from the work in [14] to achieve both high performance and robustness to prediction errors by increasing and then decreasing chunk sizes throughout execution.

This paper is organized as follows. In Section 2 we discuss relevant related work. Section 3 briefly summarizes our application and platform model as well as our previous work on UMR. Section 4 presents the RUMR algorithm, which is evaluated in simulation in Section 5. Section 6 concludes the paper and discusses future work.

2 Related Work

A number of multi-round scheduling algorithms for divisible workloads have been proposed with the assumption that performance predictions are perfectly accurate. Most of these work assume that the amount of data to be sent for a chunk is proportional to the chunk size. The work in [18] presents a "multi-installment" algorithm that uses increasing chunk sizes throughout application execution to minimize makespan. Although this approach provides an optimal schedule for a given number of rounds, it has the following limitations: latencies associated with resource utilization are not modeled; and there is no way to determine the optimal number of rounds. Our recent work in [17, 13] addresses both these limitations. By imposing the restriction that equal sized chunks are sent to workers within a round, the UMR algorithm makes it possible to compute an optimal number of rounds while modeling resource latencies. In this work we extend UMR to account for performance prediction errors. Several works aim at maximizing the steady-state performance of very long-running applications [4, 19]. The goal is not to minimize application makespan but to obtain asymptotically optimal schedules. Note that in these works it is possible to adapt to fluctuating performance characteristics of the underlying resources as the optimal schedule is periodic and can thus be changed from one period to the next.

Multi-round scheduling for divisible workloads has also been studied assuming non-zero performance prediction errors. The algorithms in [14, 15] start application execution with large chunks and decrease chunk sizes throughout. Assuming uncertainties on task execution times, this ensures that the last chunks will not experience large absolute uncertainty. These works assume a fixed network overhead to dispatch chunks of any sizes. By contrast, we assume that the amount of data to be sent for a chunk is proportional to the chunk size, which is more realistic for most applications. With this assumption, starting by sending a large chunk to the first worker would cause all the remaining workers to be idle during that potentially long data transfer. Nevertheless, in this paper we use the fundamental ideas in [14] to extend our previous work on the UMR algorithm.

The notion of scheduling applications by combining a performance-oriented and a robustness-oriented approach is not new and has been explored for instance in [20], which uses both static scheduling and self-scheduling. Our approach is more performance efficient because it achieves better overlap of computation and communication and leverage the work in [14] for improved robustness to uncertainty.

3 Background

3.1 Application and Platform Models

We consider applications that consist of a continuously divisible workload, W_{total}, and we assume that the amount of application data needed for processing a chunk is *proportional* to the amount of computation for that chunk. As done in most previous work, we only consider transfer of application input data. The works in [11, 12] take into account output data transfers but use a single round of work allocation. Similarly, the work in [4] models output but considers only steady-state performance.

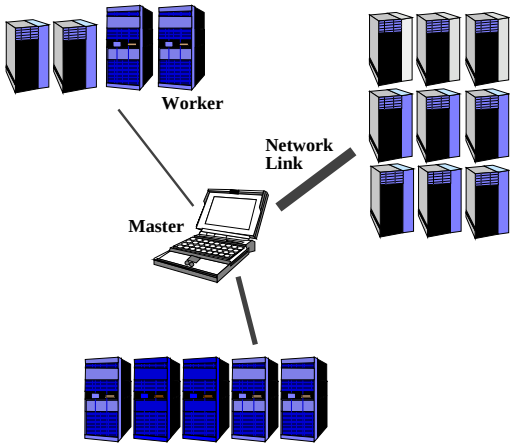


Figure 1. Computing platform model.

We assume a master-worker model with N worker processes running on N processors. We assume that the master

does not send chunks to workers simultaneously, although some pipelining of communication can occur [11]. Although this is a common assumption in most previous work, it could be beneficial to allow for simultaneous transfers for better throughput in some cases (e.g. WANs). We have provided an initial investigation of this issue in [17] and leave a more complete study for future work. The effective platform topology can then be viewed as heterogeneous processors connected to a master by heterogeneous network links (see Figure 1). Finally, we assume that workers can receive data from the network and perform computation simultaneously (as for the "with front-end" model in [21]).

Consider a portion of the total workload, $chunk \leq W_{total}$, 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

$$Tcomp_i = cLat_i + \frac{chunk_i}{S_i}, \quad (1)$$

where $cLat_i$ is a fixed overhead, in seconds, for starting a computation, and S_i is the computational speed of the worker in units of workload performed per second. Computation, including the $cLat_i$ overhead, can be overlapped with communication. We model the time spent for the master to send $chunk$ units of workload to worker i, $Tcomm_i$, as:

$$Tcomm_i = nLat_i + \frac{chunk}{B_i} + tLat_i, \quad (2)$$

where $nLat_i$ is the overhead, in seconds, incurred by the master to initiate a data transfer to worker i (i.e. initiate a TCP connection); B_i is the data transfer rate to worker i, in units of workload per second; $tLat_i$ is the time interval between when the master finishes pushing data on the network to worker i and the time when worker i receives the last byte of data. We assume that the $nLat_i + chunk/B_i$ portion of the transfer is not overlappable with other data transfer. However, $tLat_i$ is overlappable as in [11]. Note that for cases that the needed data files are replicated or pre-staged on workers, we can model these cases by using an appropriately large or infinitely large B_i. This network model is depicted on Figure 2.

This model was discussed in detail in [17, 13]. The key point is that it is flexible and can be instantiated to model platforms considered in most relevant previous works on divisible workload scheduling [11, 7, 18]. Based on our experience with actual software [22], we found that the computational latency, $cLat$, is fundamental for realistic modeling. We know of only one work that models this latency in the context of divisible load scheduling [23], but that work is only in the context of one-round scheduling algorithms.

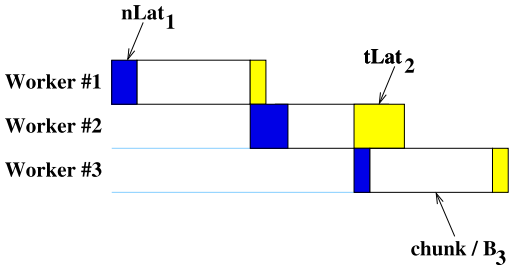


Figure 2. Illustration of the network communication model for 3 identical chunks sent to 3 workers with different values of $nLat_i$, B_i, and $tLat_i$.

3.2 The UMR Algorithm

In this section we provide a brief summary of the work and results in [17, 13] to set the stage for the RUMR algorithm, which we presented in Section 4.

Figure 3 shows how UMR dispatches chunks of workloads in multiple rounds. While this is similar in spirit to the "multi-installment" algorithm[18], UMR keeps chunk sizes *fixed* within each round. The chunk size is increased between rounds in order to reduce the overhead of starting communication ($nLat$) and computation ($cLat$). While our work in [17, 13] addresses heterogeneous platforms, but we only discuss the homogeneous case here for simplicity. The unknowns that UMR must determine are M, the number of rounds, and $chunk_j$, $j = 0, ..., M-1$, the chunk size used at each round.

Our development of UMR was as follows. We first obtained a simple induction relation on the chunk sizes, meaning that the only chunk size that needs to be determined is $chunk_0$. Then, we framed the scheduling problem as a constrained optimization problem: the goal being to minimize the application execution time subject to the constraint that all the chunks sum up to the total workload. Using the Lagrange Multiplier method [24] we obtained a system of 2 equations with M and $chunk_0$ as unknown. This system can be solved numerically by bisection (requiring about 0.07 seconds on a 400MHz PIII). Complete details are provided in [17].

Our main contribution is that we were able to compute an approximately optimal number of rounds while using a realistic platform model incorporating resource latencies. To evaluate the effectiveness of our approach we used simulation and compared UMR with the multi-round algorithm in [18] and the one-round algorithm in [11] for an extensive space of platform configurations and capabilities. We found that:

1. UMR leads to better schedules than its two competi-

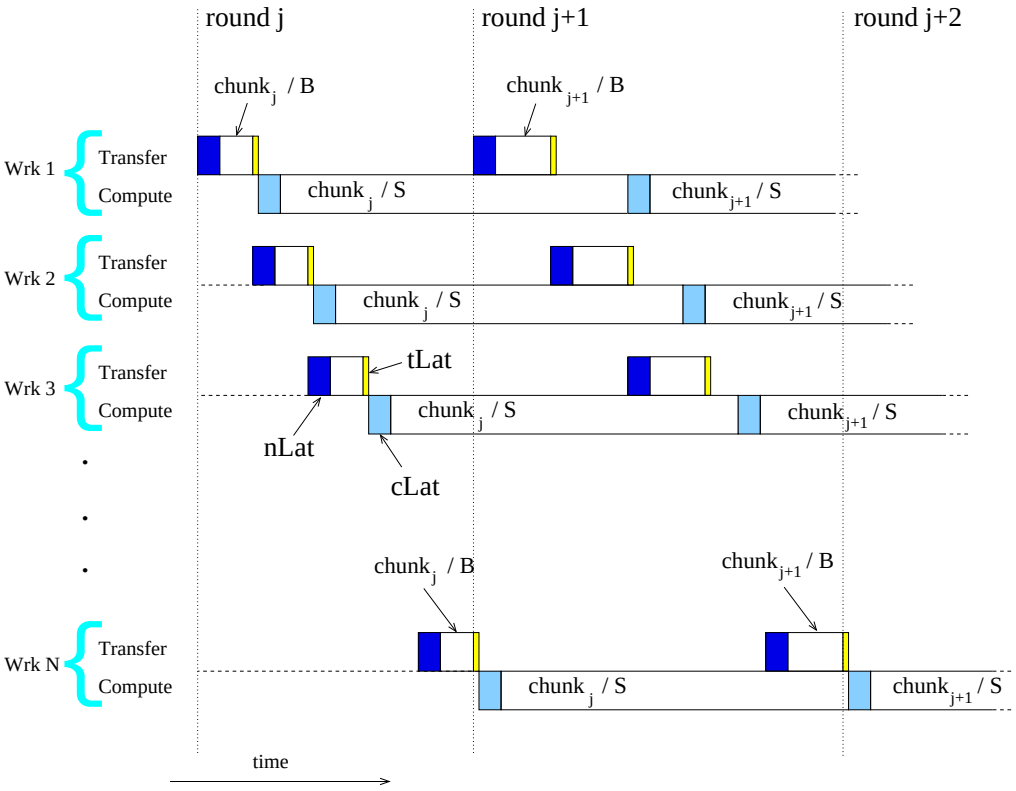


Figure 3. UMR dispatches the workload in rounds, where the chunk size if fixed within a round, and increases between rounds.

tors in an overwhelming majority of the cases in our experiments (>95%);

2. When UMR is outperformed, it is very close to the competitors (on average within 2.04% with a standard deviation of 0.035); and
3. Neither competitor ever outperforms UMR "across the board" (i.e. for ranges of computation/communication ratios).

UMR is able to achieve such improvement over previous work in spite of the uniform round restriction, an precisely because this restriction makes it possible to compute an optimal number of rounds. This is one of the main results of our previous work. We also showed that UMR tolerates high platform heterogeneity due to an effective resource selection technique. In this paper we extend UMR to account for performance prediction errors.

4 RUMR: Robust Uniform Multi-Round

The duration of a computation or of a data transfer often cannot be predicted perfectly accurately in practice. Prediction errors arise due to uncertainties of both the platform and the application. On a non-dedicated platform it is almost impossible to make accurate predictions due to resource load fluctuations (both on CPUs and network). Prediction errors may also be caused by inherent properties of the application if computation is data-dependent, i.e. the amount of computation required for a chunk depends on the nature of the data for the chunk. For instance, in a raytracing application the time taken to trace through one pixel depends greatly on the complexity of the scene. As a result, an approach in which the entire schedule is *precalculated* at the onset of the application [18, 13] is likely to be inefficient. Nevertheless, this is a standard approach in the scheduling literature, and in particular for developing divisible workload algorithms that use increasing chunk sizes [18], including our own work on UMR.

On the other extreme, algorithms particularly targeted at tolerating prediction errors do not make use of performance predictions at all [14, 15]. They exponentially decrease chunk sizes and schedule chunks in a greedy fashion. One issue there is the overhead for scheduling small chunks, which is addressed in [15]. More importantly, these algorithms do not achieve good overlap of communication and computation, which is critical for high performance.

Our basic approach is to combine both approaches:

RUMR schedules the workload in **two consecutive phases**: Phase #1 uses a revised version of UMR to pre-calculate the initial portion of the schedule, first using small chunk sizes and gradually increasing chunk sizes; Phase #2 uses the factoring approach in [14] to decrease chunk sizes. Phase #1 aims for high performance via efficient communication computation overlap and overhead reduction, while phase #2 limits the negative effect of performance prediction errors at the end of execution. In what follows we describe a model for these errors and our key design choices for RUMR.

4.1 Performance Prediction Error Model

We assume a simple prediction error model both for data transfers and computations: the ratio of predicted execution time to effective execution time is normally distributed with mean 1 and standard deviation $error$ (the distribution is truncated to avoid negative values). This model is quite general and was used in the relevant previous literature [14, 15]. Its simplicity makes it straightforward to interpret simulation results. Some of our intuitions for developing RUMR are based on the assumption of normally distributed errors (as it was done in [14, 15]). We also assume that the probability distribution of prediction errors is stationary throughout the application run. If it is not stationary but does not change too rapidly, our approach should still be effective as phase #2 does not use prediction errors at all. We also ran all the experiments under a uniformly distributed error model, but our results were essentially similar. We leave more sophisticated and/or realistic error models for future work.

A key question is whether $error$ is a known quantity, i.e. whether RUMR can use its value to decide on how to organize the schedule at the onset of the application. Estimations of $error$ can be obtained by past experience with the application and the platform, by querying resource monitoring/forecasting services [25], by monitoring prediction errors as the application runs, or by any combination of these. In what follows we discuss alternate strategies whether $error$ is known or unknown.

4.2 Design Choices for RUMR

We faced three issues for implementing RUMR:

(i) When should RUMR switch to phase #2?
(ii) What about prediction errors in phase #1?
(iii) What should the minimal chunk size be in phase #2?

We have made pragmatic design choices to address each of these questions. We provide here intuition for these choices and will validate them in the next section.

To address question (i) we use the following commonsense observation: the higher the prediction errors, the larger the proportion of the workload that should be scheduled in phase #2. If $error$ is zero then RUMR defaults to UMR and uses only phase #1. If $error$ is greater than 1, then RUMR defaults to Factoring and uses only phase #2. Otherwise, we use the following heuristic: RUMR schedules $error \times W_{total}$ units of workload in phase #2. There is one added constraint: if $error \times W_{total}/N < \times(cLat + nLat \times N)$, then RUMR does not use phase #2. The right side of the inequality is the amount of overhead incurred to send out a round of empty chunks (non-hidden latencies to send N messages and start the computation for the last processor). The lest side of the inequality is the amount of work to be accomplished in phase #2 for one worker. Our rationale is that the time to process the remaining application workload in phase #2 should be at least equal to the overhead. If $error$ is unknown, then one can just pick an arbitrary point at which to switch to phase #2, which we discuss further in Section 5.2.1.

For question (ii), it is important that the schedule do not suffer too many gaps, meaning that processors that finish computing earlier than expected should not stay idle. RUMR uses a very simple approach: send a new chunk of data to a worker if it finishes prematurely, which is in some sense incorporating a greedy scheduling component into UMR, while preserving the increasing chunk size property.

Question (iii) arises because Factoring reduces chunk sizes exponentially. We must ensure that the last chunks being sent are not so small as to incur prohibitive overhead. As observed in our answer to question (i), the overhead incurred to send one round of empty chunks is $(cLat + nLat \times N)$. If $error$ is known then we bound chunk sizes below by $(cLat + nLat \times N)/error$. Otherwise, we use $(clat + nlat \times N)$, as done in [15].

5 Simulation Results

In the previous section we have outlined the basic principles for RUMR and provided intuitive justifications for our design choices. In this section we present experimental results obtained in simulation with the goals of (i) comparing RUMR to previously proposed algorithms; and (ii) quantifying the impact and effectiveness of our design choices. To this end we have built a simulator based on the SIMGRID toolkit [26, 27]. Although our approach, like the UMR approach, was developed both for homogeneous and heterogeneous platforms, we only present results for the homogeneous case. This is for several reasons: the results are more straightforward to understand and compare; some of the competing algorithms are not amenable to heterogeneous platforms; and the purpose of our evaluation is primarily

Parameter	Values
Number of processors	$N = 10, 15, 20, \ldots, 50$
Workload (unit)	$W_{total} = 1000$
Compute rate (unit/s)	$S = 1$
Transfer rate (unit/s)	$B = (1.2, 1.3, \ldots, 2) \times N$
Computation latency (s)	$cLat = 0.0, 0.1, \ldots, 1$
Communication latency (s)	$nLat = 0.0, 0.1, \ldots, 1$

Table 1. Parameter values for the experiments presented in Section 5.

to understand the impact of performance prediction errors (see [17, 13] for a study of heterogeneity).

The parameter values we used for our experiments are shown in Table 1. The "units" in Table 1 is the minimimal unit of computation in the worklad, e.g. one sequence in a sequence datafile, or one block of pixels in image processing. Note that $B_i = B$, $cLat_i = cLat$, and $nLat_i = nLat$ for all processors and network links so that the platform is homogeneous. We use $S_i = 1$ so that the numerical value B is also the communication to computation ratio. Note that the value of B depends on N. This is to comply with the full platform utilization conditions that we developed in [17]. If these conditions are not met then the number of processors must be reduced. We do not consider this issue in this paper for simplicity.

In all experiments we vary $error$, as defined in Section 4.1, from 0.0 to 0.5. All presented results are averages obtained over 40 repetitions (to account for the randomness of prediction errors).

We assume that before the master begins sending workload, all the input data corresponding to the workload have been placed on the master. Also, we so not model background resource load explicitly. Indeed, we account for resource uncertainty with the the $error$ parameter.

5.1 Comparison with Competing Algorithms

We compared RUMR with the following three competing algorithms: the UMR algorithm (see Section 3.2), the Multi-Installment (MI) algorithm in [18], and the Factoring algorithm in [14] .The first two were designed to achieve high performance in perfectly predictable environments whereas the last one was designed to be robust to performance prediction errors. We instantiated four versions for MI, MI-x for $x = 1, \ldots, 4$, where x is the number of rounds used. This is necessary because MI, unlike UMR, does not provide a way to determine an optimal number of rounds automatically. We also investigated the Fixed-Size Chunking (FSC) strategy described in [15]. This strategy is an optimized self-scheduling algorithm and performs worse than Factoring in most of our experiments. Consequently we do not show results for FSC.

	Ranges of $error$				
Algorithm	0–0.08	0.1–0.18	0.2–0.28	0.3–0.38	0.4–0.48
UMR	54.96	56.60	73.45	81.99	86.48
MI-1	98.27	86.08	75.27	68.27	69.82
MI-2	94.44	88.38	94.95	98.91	98.61
MI-3	94.70	95.70	97.33	98.76	99.94
MI-4	95.55	97.77	98.17	98.71	99.84
Factoring	98.21	94.06	93.84	90.16	84.74

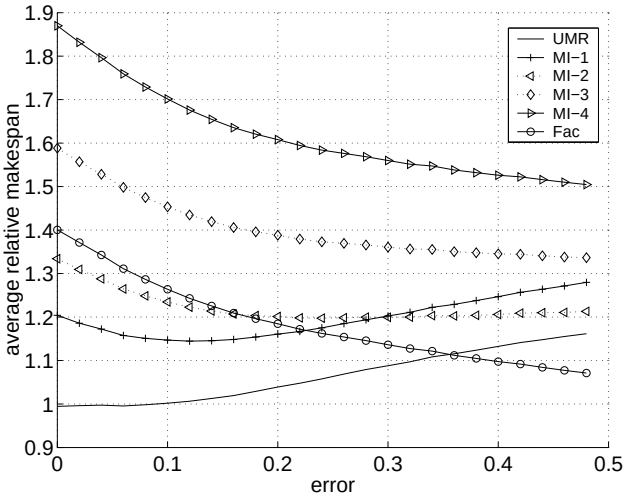
Table 2. Percentage of experiments for which RUMR outperforms the algorithms in the leftmost column.

	Ranges of $error$				
Algorithm	0–0.08	0.1–0.18	0.2–0.28	0.3–0.38	0.4–0.48
UMR	0.00	4.64	27.59	43.29	55.80
MI-1	68.89	44.97	48.70	56.25	57.02
MI-2	59.67	56.64	65.55	69.74	70.03
MI-3	69.55	68.51	85.24	90.92	93.03
MI-4	76.46	78.49	90.18	94.73	96.70
Factoring	90.09	61.88	45.62	35.39	23.86

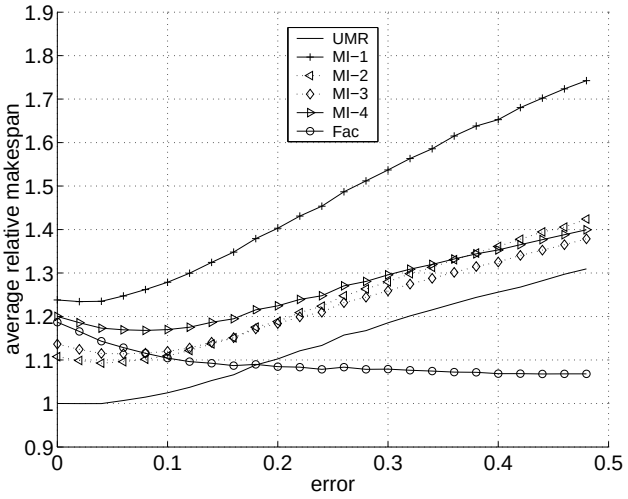
Table 3. Percentage of experiments for which RUMR outperforms the algorithms in the leftmost column by at least 10%.

To provide an overview of our results, Table 2 shows the percentage of times that the algorithms in the leftmost column are outperformed by RUMR for all our experiments, averaged over five ranges of $error$ values. These results demonstrate that RUMR outperforms competing algorithms in most of the cases as all values are above 55% and up to 99%. Overall RUMR outperforms competing algorithms in 79% of our experiments. To quantify the amount by which RUMR outperforms competitors, Table 3 show the percentage of times that UMR outperforms the algorithms in the leftmost column by at least 10%. One can see that RUMR outperforms MI-x by at least 10% in between 45% and 95% of the experiments. One can see interesting and inverted trends for UMR and Factoring as $error$ grows. While these results give a sense of the superiority of RUMR, we present below relative makespan results that provide more detailed insight.

Figure 4(a) plots the average makespan achieved by the six competing algorithms normalized to that achieved by RUMR versus $error$. These results are averages over all experiments, covering the entire space of parameter values shown in Table 1. Values above 1.0 represent cases in which RUMR outperforms the competing algorithms. We present only *relative* makespan results as experiments are conducted for a wide range of platform configurations, making absolute makespans difficult to compare and average. The only algorithm that outperforms RUMR on av-



(a) Data for all parameters in Table 1.



(b) Data for a subset of the parameters in Table 1: $cLat < 0.3$, $nLat < 0.3$.

Figure 4. Makespan obtained with UMR, MI-x, and Factoring normalized to that of RUMR versus $error$.

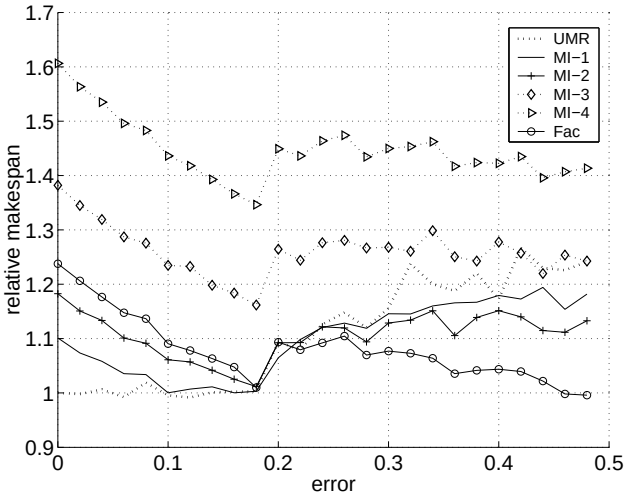


Figure 5. Makespan obtained with UMR, MI-x, and Factoring normalized to that of RUMR versus $error$, for a specific instantiations of the parameters in Table 1: $cLat = 0.3$, $nLat = 0.9$, $N = 20$, $r = 36$.

erage is UMR when the prediction error is small, which is explained in Section 5.2.2. As expected, the relative performance of Factoring improves as $error$ increases. Nevertheless RUMR outperforms Factoring by at least 8% for values of $error$ up to 0.5.

Another observation is that UMR (and thus RUMR) outperforms MI-x on average. This extends our previous results from [13] in which we only looked at the case $error = 0$. In fact The MI-x algorithms never get within less than 20% of RUMR on average. However, the trend is an initial decrease of the average relative makespan and the curves for MI-3 and MI-4 keep decreasing as $error$ grows. This phenomenon is due to the fact that in high latency situations RUMR often uses only one round in phase #1 (due to the way in which UMR operates), and does not use a phase #2 if $error \times W_{total}$ is smaller than $N \times nLat + cLat$. Therefore, it is possible that RUMR uses only a single round of work allocation and thus loses some robustness to performance prediction errors when compared to, say, MI-3 or MI-4. This is demonstrated in Figure 4(b), which is similar to Figure 4(a) but for a subset of the parameter space: $nLat < 0.3$ and $cLat < 0.3$. One can see under smaller latency situations all versions of MI-x exhibit a brief initial decrease followed by a steady increase. This increase is due to RUMR starting to deem $error$ large enough to use a phase #2, as demonstrated below.

The results we have presented so far are averages over large portions of our parameter space. When inspecting results more closely we found that results fall roughly in two

categories: large and small $nLat$ values. When $nLat$ is relatively small RUMR generally uses many rounds in phase #1 leading to good performance and relative makespan patterns are similar to those seen in Figure 4(b). When $nLat$ is relatively large then the patterns are similar to those seen in Figure 4(a), with a few interesting elements. For instance, Figure 5 shows average relative makespans for a single point in our parameter space with high $nLat = 0.9$. In this situation RUMR uses only one round in phase #1 for low values of $error$, causing the aforementioned initial decrease of the relative makespans for MI-x. However, we can see a sharp increase in relative makespans at $error = 0.18$. This corresponds to RUMR starting to use phase #2 to schedule the last portion of the workload. This pattern explicitly demonstrate the benefit of splitting the execution in two phases.

5.2 Evaluation of RUMR Design Choices

5.2.1 Division in Two Phases

We wanted to evaluate the effectiveness of our heuristic to decide when RUMR should switch to phase #2 (design choice (i) in Section 4.2). Our approach is to reserve $error \times W_{total}$ of the entire workload for scheduling in phase #2 as long as it is larger than the overhead caused by sending chunks to all workers. This assumes that the value of $error$ can be estimated. We evaluated an implementation of RUMR that schedules a fixed percentage of the workload in phase #1, independently of $error$. We used phase #1 percentage values of 50%, 60%, 70%, 80% and 90%. Figure 6 plots average makespans relative to the makespan of the original RUMR algorithm versus $error$, over all parameter values in Table 1.

One can see that when $error$ is small it is of course more beneficial to schedule a large percentage of the workload in phase #1. We can also see that the original RUMR achieves a much better makespan because it actually does not use a phase #2 at all. As $error$ gets large, versions of RUMR that schedule a larger fraction of the workload in phase #1 lead to worse performance than the original RUMR. whereas versions that use a smaller fraction get comparable performance. Overall, the version of RUMR that schedules 80% of the workload in phase #1 and 20% in phase #2 achieves the best relative performance when averaged over $error$ (within about 15% of the makespan obtained with the original RUMR). Note these curves don't neccessarily interesect the x-axis, because orginal RUMR also uses $N \times (cLat + nLat \times N)$ as a threshold for phase #2, so, at $error = 0.1$, original RUMR doesn't necessarily give 90% to phase #1, which RUMR_90 does.

From these results we conclude that our heuristic for partitioning the workload between phase #1 and phase #2 is effective across the board (i.e. for the whole range of values for $error$). This is actually not surprising, but leads to an interesting point. In cases where the value of $error$ can not be known in advance we have to use a fixed division between the two phases (80% in phase #1 seems like a good practical choice) ; But RUMR can easily achieves better performance if techniques to estimate $error$ (even coarsely) are available. In Section 4.2 we have pointed to such techniques, which we now view as critical for scheduling divisible workloads in environments that are not perfectly predictable but with quantifiable prediction error magnitude. These techniques have been used effectively in previous work [20] and we will reuse them to implement RUMR as part of a practical application execution environment [16] in future work.

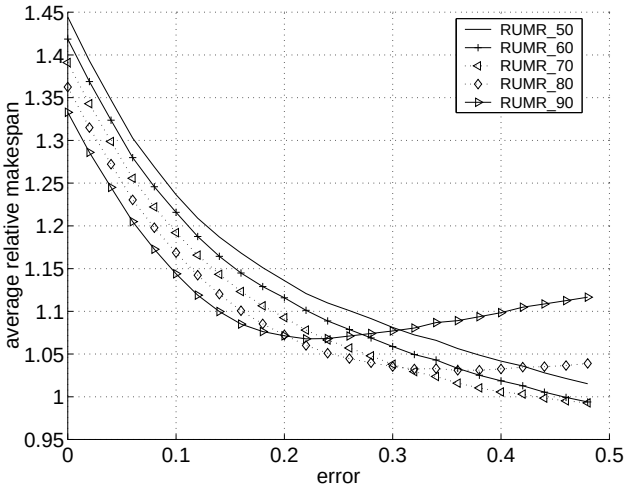


Figure 6. Average makespan obtained with RUMR when scheduling a fixed percentage of the workload in phase #1 normalized to that obtained with the original RUMR versus $error$, for all parameters in Table 1.

5.2.2 Modification to UMR in Phase #1

In Section 4.2 we mentioned that we modified UMR in phase #1 so that out-of-order chunk dispatching is allowed if processors become idle prematurely. We compared this strategy with a version of RUMR that uses a plain UMR in phase #1, i.e. with in-order chunk dispatching. Figure 7 plots the makespan achieved by the plain UMR version normalized to the makespan achieved by the original RUMR versus $error$. We see that allowing out-of-order chunk dispatching only leads to only about 1% improvement for high values of $error$. Furthermore, for very low values of $error$ using plain UMR is slightly more efficient.

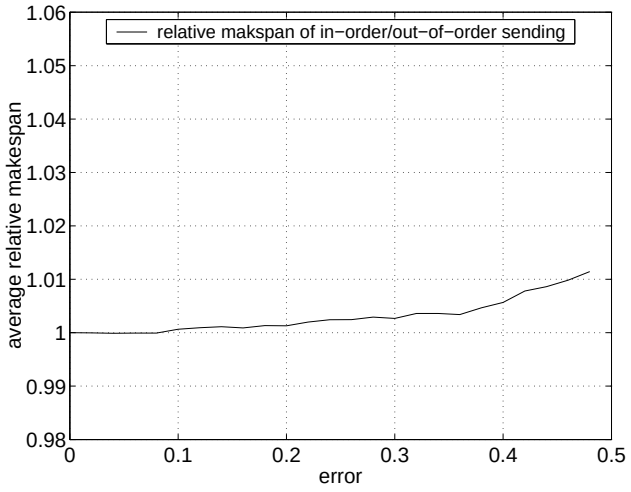


Figure 7. Average makespan obtained with RUMR when using plain UMR in phase #1 normalized to that obtained with the original RUMR versus $error$, for all parameters in Table 1.

Indeed, the makespan can be harmed by perturbations in chunk ordering that are not judicious when prediction errors are minute and would not impact the overall makespan much anyway. Although using out-of-order chunk dispatching seemed promising, these results shows that it is only marginally effective and that most of the effectiveness of RUMR comes from the division into two phases.

6 Conclusion

In this paper we have presented RUMR (Robust Uniform Multi-Round), a scheduling algorithm for minimizing the makespan of divisible workload applications under uncertainties of resource performance, our ultimate goal is to develop a scheduling strategy for divisible workloads that can be used in practice for real-world applications on real-world platforms.

In our previous work [17, 13] we made a first contribution by developing UMR (Uniform Multi-Round), an algorithm that outperforms previously proposed algorithms while tolerating more realistic latency models. In this paper we have taken the next step and addressed the issue of performance prediction errors that arise due to uncertainties about platforms and applications. RUMR leverages UMR and the work in [14] to achieve both high performance and robustness to prediction errors: it uses two consecutive phases for application execution, with increasing and then decreasing workload chunk sizes.

We have evaluated our approach with extensive simulation experiments. We have demonstrated that RUMR outperforms previously proposed algorithms both in terms of performance and robustness. We then have presented results to evaluate our design choices when implementing RUMR and in particular we showed that the way in which RUMR divides application execution in two phases is key to its effectiveness. Further simulations will be done on more complex and realistic error distribution models, and use traces from real applications.

While our work could be extended at an algorithmic level and while further optimizations can be envisioned, our next step in this research will focus primarily on integrating RUMR as part of the APST Grid application execution environment [28, 29]. We will extend APST so that it supports divisible workloads and we will implement RUMR as part of APST's scheduler. This implementation will make it possible to determine empirical performance prediction error distributions as well as latencies for sending data and starting computation as the application runs. Such information will be used on-the-fly by RUMR to make relevant scheduling decisions. Finally, we will perform large numbers of experiments (building on APST's large user and application base) to validate our approach. The end result will be a practical and usable implementation of a scheduling strategy for deploying high performance divisible workloads in real-world settings.

References

[1] Tim Davis, Alan Chalmers, and Henrik Wann Jensen. *Practical parallel processing for realistic rendering*. ACM SIGGRAPH, July 2000.

[2] BLAST Webpage. `http://http://www.ncbi.nlm.nih.gov/BLAST/`.

[3] C. Lee and M. Hamdi. Parallel Image Processing Applications on a Network of Workstations. *Parallel Computing*, 21:137–160, 1995.

[4] 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*, 1998.

[5] T. Hsu. Task Allocation on a Network of Procesors. *IEEE Transactions on Computers*, 49(12):1339–1353, december 2000.

[6] T. Braun, H. Siegel, and N. Beck. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61:810–837, 2001.

[7] O. Beaumont, A. Legrand, and Y. Robert. The Master-Slave Paradigm with Heterogeneous Processors. In *Proceedings of Cluster'2001*, pages 419–426. IEEE Press, 2001.

[8] 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.

[9] J. Blazewicz, M. Drozdowski, and M. Markiewicz. Divisible Task Scheduling - Concept and Verification. *Parallel Computing*, 25:87–98, 1999.

[10] M. Drozdowski and P. Wolniewicz. Experiments with Scheduling Divisible Tasks in Clusters of Workstations. In *Proceedings of Europar'2000*, pages 311–319, 2000.

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

[12] D. Altilar and Y. Paker. Optimal Scheduling algorithms for Communication Constrained Parallel Processing. In *Proceedings of Europar'02*, pages 197–206, 2002.

[13] 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'03), Nice, France*, April 2003. to appear.

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

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

[16] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of Supercomputing'00*, November 2000.

[17] 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.

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

[19] O. Beaumont, A. Legrand, and Y. Robert. Optimal algorithms for scheduling divisible workloads on heterogeneous systems. Technical Report RR-2002-36, Laboratoire de l'Informatique du Parallélism, École Normale Supérieure de Lyon, France, October 2002.

[20] N. Spring and R. Wolski. Application Level Scheduling of Gene Sequence Comparison on Metacomputers. In *Proceedings of the 12th ACM International Conference on Supercomputing, Melbourne, Australia*, July 1998.

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

[22] H. Casanova and F. Berman. *Parameter Sweeps on the Grid with APST*, chapter 26. Wiley Publisher, Inc., 2002. F. Berman, G. Fox, and T. Hey, editors.

[23] B. Veeravalli, X. Li, and C. Ko. On the Influence of Start-Up Costs in Scheduling Divisible Loads on Bus Networks. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1288–1405, Dec. 2000.

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

[25] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1:119–132, January 1998.

[26] 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), Tokyo, Japan*, May 2003. to appear.

[27] SIMGRID Webpage. `http://grail.sdsc.edu/projects/simgrid`.

[28] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of Supercomputing 2000 (SC'00)*, Nov. 2000.

[29] APST Webpage. `http://grail.sdsc.edu/projects/apst`.