UMR: A Multi-Round Algorithm for Scheduling 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

Abstract

In this paper we present an algorithm for scheduling parallel applications that consist of a divisible workload. Our algorithm uses multiple rounds to overlap communication and computation between a master and several workers. Multi-round scheduling has been used for divisible workloads in previous work and our contribution is as follows. We use "uniform" rounds, i.e. a fixed amount of work is sent out to all workers at each round. This restriction makes it possible to compute an approximately optimal number of rounds, which was not possible for previously proposed algorithms. In addition, we use more realistic platform models than those used in previous works. We provide an analysis of our algorithm both for homogeneous and heterogeneous platforms and present simulation results to quantify the benefits of our approach.

1 Introduction

Applications that consist of many independent computational tasks arise in many fields of science and engineering [1, 2, 3, 4]. These applications often require large amounts of compute resources as users wish to tackle increasingly complex problems. Fortunately, advances in commodity technology (CPU, network, RAM) have made clusters of PCs cost-effective parallel computing platforms. In this paper we address the problem of scheduling the aforementioned applications on such platforms with the goal of reducing execution time, or *makespan*. This problem has been studied for two different application models: *fixed-sized tasks* and *divisible workload*. In the first scenario, the application's workload consists of a number of tasks whose size are pre-determined and a number of scheduling heuristics have been developed [5, 6, 7, 8]. In the divisible workload scenario, the scheduler can partition the workload in arbitrary "chunks". The usual assumption is that the workload is continuous. In practical situations, this often means that the execution time of a base computational unit is orders of magnitudes smaller than the execution time of the entire workload and that all base computational units are the same size. In this paper we focus solely on the divisible workload scenario, which has been extensively studied [9].

The divisible workload scheduling problem is challenging due to the overhead involved when starting tasks: (i) the time to transfer application input/output data to/from a compute resource; (ii) the latency involved in starting a computation. In [9], the problem is identified as: *Given an arbitrarily divisible workload ... in what proportion should the workload be partitioned and distributed among the processors so that the entire workload is processed in the shortest possible time?* The trade-off for achieving a good schedule is as follows. On the one hand, dividing the workload into large chunks generally reduces the overhead, and thereby the execution time of the application. On the other hand, dividing the workload into small chunks makes it possible to overlap overhead with useful work more efficiently. In all that follows we consider a traditional *master/worker* paradigm.

Our contributions in this paper are on several fronts. We propose and analyze a new scheduling algorithm: UMR (Uniform Multi-Round). Similarly to previously proposed algorithms, UMR dispatches work to compute resources in multiple rounds. However, we add the restriction that rounds must be "uniform", i.e. within each round the master dispatches identical chunks to all workers. Due to this restriction, we are able to derive an approximately optimal number of rounds, both for homogeneous and heterogeneous platforms. We evaluate our algorithm with models that are more realistic than those used in previous work. We compare our algorithm with a previously proposed multi-round algorithm and a one-round algorithm. Our simulation results demonstrate the benefits of our approach for wide ranges of scenarios. We also analyze the impact of various system parameters on the behavior and effectiveness of UMR.

2 Related Work

The works in [10, 11, 12, 3, 13] study scenarios in which the workload is divided in as many chunks as processors. Therefore, the entire application is performed in a single round of work allocation. By contrast, our algorithm uses multiple rounds. During each round a portion of the entire workload is partitioned among the available processors. Therefore, our work is most related to the "multi-installment" algorithm presented in [14]. The key idea is that using small chunks and multiple rounds allows for overlapping of communication and computation. Note that in both our work and the work in [14] it is assumed that the amount of data to be sent for a chunk is proportional to the chunk size.

The chunk size can then be gradually increased throughout the application run in order to reduce communication overhead. Our approach differs from [14] in the following way. Whereas [14] allocates decreasing chunks of the workload to processors within a round, we keep the chunk size *fixed within a round*. This has one major benefit: our algorithm is amenable to analysis, which allows us to compute a near-optimal number of rounds, both for homogeneous and heterogeneous platforms. We provide quantitative comparison between our work and [14] in section 5.1. The work in [4] uses multiple rounds to schedule divisible workloads. However, it focuses on *steady-state* application performance rather than makespan and therefore uses identical rounds.

Multi-round scheduling for divisible workloads has also been studied in [15, 16]. Instead of increasing chunk size throughout application execution, those approaches start with large chunks and *decrease* chunk size throughout application execution. Assuming uncertainties on task execution times, reducing the chunk size ensures that the last chunks will not experience large absolute uncertainty. The work in [15, 16] assumes a fixed network overhead to dispatch chunks of any sizes. 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. In this paper we do not consider task execution time uncertainties, but discuss possible ways in which our work can leverage that in [15, 16].

3 Models

3.1 Application

We consider applications that consist of a workload, W_{total}, that is *continuously divisible*: the scheduler can decide how big a chunk of the workload to give out to a processor. 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 [12, 13] 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.

3.2 Computing Platform

We assume a *master/worker* model with N worker processes running on N processors. The master sends out chunks to workers over a network. We assume that the master uses its network connection in an sequential fashion: it does not send chunks to workers simultaneously, even though some pipelining of communication can occur [12]. This is a common assumption and is justified either by the master's implementation, or by the properties of the network links (e.g. a LAN). In some cases, for instance on a WAN, it would be beneficial for the master to send data to workers simultaneously in order to achieve better throughput. We leave this issue for future work. Note that we do not require that the speeds of network communications to each worker be identical. Therefore, the platform topology consists of network links with various characteristics to sets of heterogeneous processors, as depicted in 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 [9]).

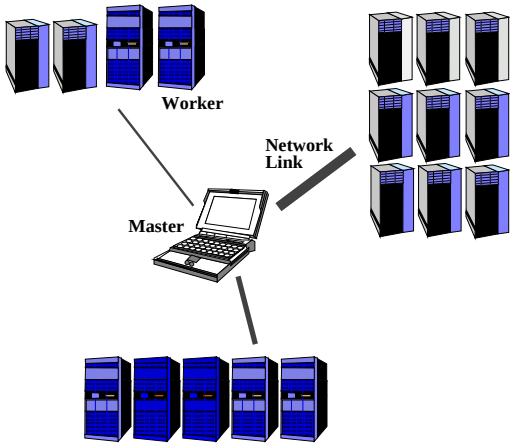

Figure 1. Computing platform model.

Let us formalize our model. 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 (e.g. for starting a remote process), 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 (e.g. pre-process application input data and/or 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 (to model pipelined networking as in [12]). This model is depicted on Figure 2 for data transfers from the master to 3 workers.

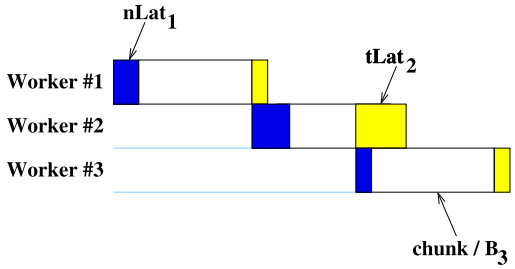

Figure 2. Illustration of the network communication model for 3 identical chunks sent to 3 workers with different values of $nLat_i$, B_i, and $tLat_i$.

This model is flexible enough that it can be instantiated to model several types of network connections. For instance, setting the $nLat$ values to 0 models a pipelined network such as the one used in [12]. In that case, the $tLat$ values represent the network latency between the master and the workers. The model can also be instantiated with non-zero $nLat$ values and zero $tLat$ values as in [7]. This is representative of distinct connections being established for each individual transfer, with no pipelining. Zero $nLat$ and zero $tLat$ corresponds to the work in [14]. To the best of our knowledge, no other work models computation latency, $cLat$. Based on our experience with actual software [17], we deem $cLat$ to be fundamental for realistic modeling. We provide an analysis of our scheduling algorithm using this generic platform model, and thereby validate our approach for a broad range of platforms.

4 The UMR Algorithm

Similar to the algorithm presented in [14], UMR dispatches chunks of the workload in rounds. The chunk size is increased between rounds in order to reduce the overhead of starting communication ($nLat$) and computation ($cLat$). Unlike [14], we keep the chunk size fixed within each round. We are able to compute near-optimal number of rounds, and a near-optimal chunk size at each round, as demonstrated in the next section. In all that follows, M denotes the number of rounds used by UMR.

4.1 UMR on Homogeneous Platforms

We describe and analyze the UMR algorithm for a platform that consists of N *identical* workers accessible via one network link. Consequently we set:

$$\forall i = 1, .., N$$
$$S_i = S,\ nLat_i = nLat,\ tLat_i = tLat,\ B_i = B \quad (3)$$

Induction on chunk sizes – Let $chunk_j$, for $j = 0, .., M-1$, be the chunk size at each round. We illustrate the operation of UMR in Figure 3. At time T_A, the master starts dispatching chunks of size $chunk_{j+1}$ for round $j+1$. The workers perform computations of sizes $chunk_j$ for round j concurrently. To maximize bandwidth utilization, the master must finish sending work for round $(j+1)$ to all workers before worker N finished its computation for round j, which is shown at time T_B. Therefore, perfect bandwidth utilization is achieved when:

$$tLat + cLat + \frac{chunk_j}{S} = N(\frac{chunk_{j+1}}{B} + nLat) + tLat. \quad (4)$$

The left-hand side is the time worker N spends receiving the last bytes of data, initiating a computation, and computing a chunk during round j. The right-hand side is the time it takes for the master to send data to all N workers during round $j+1$. Eq. 4 defines a simple induction for $chunk_j$, and one can then compute:

$$\begin{aligned} & \forall j \quad chunk_{j+1} - \alpha = \frac{B}{NS}(chunk_j - \alpha) \\ \Rightarrow \quad & \forall j \quad chunk_j = (\frac{B}{NS})^j (chunk_0 - \alpha) + \alpha, \end{aligned} \quad (5)$$

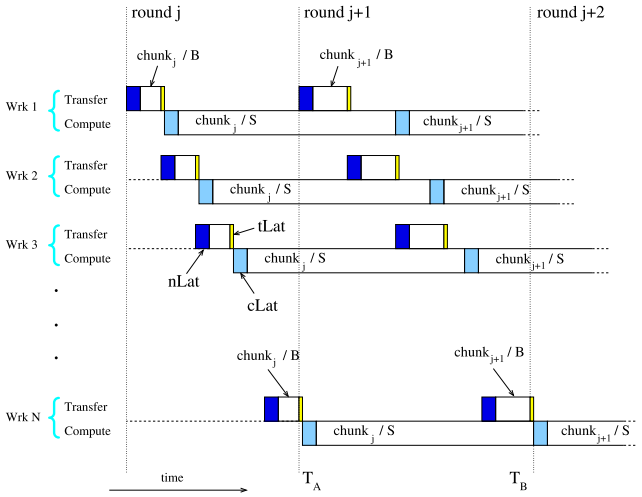

Figure 3. UMR dispatches the workload in rounds, where the chunk size if fixed within a round, and increases between rounds.

where

$$\alpha = \frac{BS}{B - NS}(N \times nLat - cLat). \tag{6}$$

We have thus obtained a geometric series of chunk sizes, where $chunk_0$ is an unknown.

Necessary conditions for full platform utilization – Let us determine the conditions under which all N workers can be utilized. To utilize all workers, the master must be able to send out all work for round j and work for round $j+1$ to worker #1 before this worker becomes idle. This can be written formally as:

$$\left[N(nLat + \frac{chunk_j}{B}) + nLat\right] + \left[\frac{chunk_{j+1}}{B} + tLat\right] \leq \left[nLat + \frac{chunk_j}{B} + tLat\right] + \left[cLat + \frac{chunk_j}{S}\right],$$

where the left-hand side is the time needed by the master to send all data for round j and data to worker #1 for round $j+1$, and the right-hand side is the time worker #1 spends receiving data and computing for round j. Replacing $chunk_{j+1}$ by its expression given in Eq. 5, we obtain:

$$(NS - B)chunk_0 \leq (NS - B)\alpha. \tag{7}$$

If this constraint it not met, then at least one worker can not be used and N should be reduced. We will see in what follows how $chunk_0$ is computed by our algorithm. We give here necessary conditions for Eq. 7, which are analogous to the "$m\sigma \leq 1$" constraint in [14].

1. If $NS - B > 0$, then Eq. 7 reduces to $chunk_0 \leq \alpha$. Since $chunk_0$ must be strictly positive, a necessary condition for Eq. 7 is:
$$\alpha > 0. \tag{8}$$
2. If $NS - B < 0$, then Eq. 7 reduces to $chunk_0 \geq \alpha$. For all workers to be used, $chunk_0$ must be smaller than W_{total}/N. Therefore, a necessary condition is for Eq. 7 is:
$$\alpha \leq W_{total}/N. \tag{9}$$

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

$$Ex(M, chunk_0) = \frac{W_{total}}{N} + M \times cLat + \frac{1}{2} \times N(nLat + \frac{chunk_0}{B}) + tLat. \tag{10}$$

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 correspond to the time for the master to send all the data for round 0. The $\frac{1}{2}$ factor is due to an optimization that is described in detail in section 4.3. Finally, the fourth term, $tLat$, can be seen on Figure 3 just after time T_A for worker N.

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

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

This constrained minimization problem, with M and $chunk_0$ as unknowns, can be solved by using the Lagrange Multiplier method [18]. 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:

$$\begin{cases} \frac{\partial L}{\partial \lambda} = G = 0 \\ \frac{\partial L}{\partial M} = \frac{\partial Ex}{\partial M} + \lambda \times \frac{\partial G}{\partial M} = 0 \\ \frac{\partial L}{\partial chunk_0} = \frac{\partial Ex}{\partial chunk_0} + \lambda \times \frac{\partial G}{\partial chunk_0} = 0. \end{cases} \tag{11}$$

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

$$N\alpha - \frac{W_{total} - NM\alpha}{1 - (\frac{B}{NS})^M}\left(\frac{B}{NS}\right)^M \ln\left(\frac{B}{NS}\right) - 2cLat \times B\frac{1 - (\frac{B}{NS})^M}{1 - \frac{B}{NS}} = 0 \tag{12}$$

This equation can be solved numerically by bisection. The solve is fast (on the order of 0.07 seconds on a 400MHz PIII) and can thus be implemented in a runtime scheduler with negligible overhead. Once we have computed M^*, the solution of Eq. 12, $chunk_0$ follows as:

$$chunk_0 = \frac{(1 - \frac{B}{NS})(W_{total} - NM^*\alpha)}{N \times (1 - (\frac{B}{NS})^{M^*})} + \alpha, \qquad (13)$$

and $chunk_j (j > 0)$ can be computed with Eq. 5. Complete details on these derivations are provided in a technical report [19].

4.2 UMR on Heterogeneous Platforms

The analysis of UMR in the heterogeneous case, i.e. without the simplifying assumptions in Eq. 3, is more involved than that for the homogeneous case but follows the same steps. Due to space constraints we can not present the entire development and refer the reader to [19] for details. Nevertheless, we describe two key differences with the homogeneous case:

1. In the homogeneous case, we fixed the *size* of chunks for an round. Here, we fix the *time* it takes for each worker to perform computation during a round. In other words, worker i receives a chunk size $chunk_{ji}$ at round j, but the quantity $chunk_{ji}/S_i$ depends only on j. This makes it possible to obtain an induction on chunk sizes, necessary conditions for full platform utilization, and a constrained minimization problem that are analogous to those in the homogeneous case.

2. We have given necessary conditions for a homogeneous platform to be fully utilized. One can reduce N to meet these conditions and then iteratively reduce its value until the constrained minimization problem has a solution. Similar considerations hold for the heterogeneous case. However, there is an additional **resource selection** issue. When the full platform cannot be utilized one must select which processors not to use. In section 5.2, we present a resource selection strategy that works well in practice.

4.3 Practical Implementation of UMR

Before presenting experimental results, we present here two technical modifications of the UMR algorithm that are used in the rest of the paper.

Rounding M^* to an integer – The bisection solve of Eq. 12 produces a real value for M^*, whereas we need an integral number of rounds. A possibility is to use $\lceil M^* \rceil$ rounds. The last round would then consist in dispatching potentially small amounts of work equal to $chunk_{\lceil M^* \rceil}$, while still incurring full $cLat$ overheads. Instead, we use the value $\lfloor M^* + \frac{1}{2} \rfloor$ as the number of rounds, which works better in practice.

Last round optimization – The work in [14] shows that in an optimal divisible workload schedule all workers finish computing at the same time. In the UMR algorithm, as it described in section 4, the finishing time of all N workers has the same "slope" as the starting of the compute times in the first round (as seen in Figure 3). When communications are relatively slow, i.e. when B/S is low, worker 1 finishes computation much earlier than worker N, leading to idle time. To alleviate this limitation, we modify the implementation of UMR for the last round. The main idea is to give a decreasing amount of work to workers during the last round in order to have them all finish at the same time (note that this is similar to what is done in [14] and is different from the uniform round approach we use for all other rounds). The straightforward computation of the modified chunk sizes for the last round is presented in [19]. This modification of the last round leads to the $\frac{1}{2}$ factor in Eq. 10.

5 Simulation Results

In order to evaluate our approach, we developed a simulator with the Simgrid [20] toolkit. First, we compared UMR to previously proposed algorithms: the multi-round algorithm in [14] and the one-round algorithm in [12]. Second, we evaluate UMR's robustness to platform heterogeneity. Additionally, an experimental study of how system parameters impact UMR's choice for the number of rounds is presented in [19].

5.1 Comparison with Previous Algorithms

Using our simulator, we compared UMR with the multi-installment algorithm proposed in [14], heretofore referred to as MI. Since a closed form solution for MI is not available for heterogeneous platforms we only present results for homogeneous platforms in this section. Furthermore, unlike UMR, the MI algorithm does no compute an optimal number of rounds. In fact, since the work in [14] does not model latencies, it would seem that the best scheduling strategy is to use as many rounds as possible. Of course, the authors state that in a practical scenario it would not be beneficial to use a large number of rounds (due to latencies). Consequently, we present results for the MI algorithm with 1 to 8 installments. We denote each version by MI-x with $x = 1, \ldots, 8$. We also compare UMR against the one-round algorithm in [12], which we denote as One-Batch. Unlike [12], we model only transfer of input data to the

workers. Our version of One-Batch takes into account all the latencies in our model.

We performed experiments for wide ranges of values for parameters defining the platform and the application. We first present aggregate results averaged over large numbers of experiments. We then present results for sub-sets of the results to illuminate the behaviors of the different scheduling algorithms.

5.1.1 Aggregate Results

We evaluated UMR, MI-x, and One-Batch for the parameter values in Table 5.1.1. Note that we choose $S = 1$ to limit the number of parameters. In these conditions, the computation/communication ratio for all workers is exactly equal to the numerical value of B. In all that follows, we use the terms "computation/communication ratio" and "B" interchangeably. Since the effect of $tLat$ is just to shift the running time by $tLat$, we set $tLat = 0$. Finally, note that we choose values of B that make it possible to use all workers given the necessary conditions developed in section 4.1.

For each instantiation of these parameters we did the following. We simulated all 10 scheduling algorithms, and computed three metrics for each algorithm: (i) its *makespan*, normalized to that achieved by UMR in this experiment; (ii) its *rank*, which goes from 0 (best) to 10 (worst); (iii) its *degradation from best*, which measures the percent relative difference between the makespan achieved by this algorithm and the makespan achieved by the best algorithm for this experiment. These three metrics are commonly used in the literature for comparing scheduling algorithms. We present averages of these 3 metrics for each algorithm over all parameter configurations in Table 2(a).

The main observation from Table 2(a) is that UMR outperforms competing algorithms in most cases. We also see that the One-Batch strategy outperforms the MI-x algorithm in the majority of the cases. On average it leads to schedules 4% longer than UMR. This is due to the fact that a one-round algorithm can not overlap communication and computation as much as a multi-round approach. Over all instantiations of system parameters UMR is not the best algorithm in only 4.46% of the cases. When UMR is outperformed, it is on average within 2.04% of competing algorithms with a standard deviation of 0.035.

MI-x does not take into account latencies, which explains why its performance is rather poor in our experiments. Consequently, we show results for a subset of the parameter space in Table 2(b). In this table we limit $nLat$ and $cLat$ to be below 0.1 seconds. We see that MI-2 and MI-3 perform better for the limited set of parameters: they lead to makespans that are within 6% and 9% of that achieved my UMR. Nevertheless, UMR leads to better performance in 80% of the time. Note that the One-Batch algorithm per-

Parameter	Values
# of processors	$N = 10, 15, 20, \ldots, 50$
Workload (unit)	$W_{total} = 1000$
Compute rate (unit/s)	$S = 1$
Transfer rate (unit/s)	$B = 1.1N, 1.1N + 1, \ldots, 5.0N$
Comp. latency (s)	$cLat = 0.00, 0.03, \ldots, 0.99$
Comm. latency (s)	$nLat = 0.00, 0.03, \ldots, 0.99$

Table 1. Parameter values for the experiments presented in section 5.1.

Algorithm	normalized makespan	rank	degradation from best
UMR	1.00	0.09	0.09
MI-1	1.21	2.75	21.50
MI-2	1.48	2.73	48.33
MI-3	1.84	3.68	84.48
MI-4	2.22	4.74	122.09
MI-5	2.60	5.79	160.04
MI-6	2.98	6.83	198.11
MI-7	3.36	7.85	236.22
MI-8	3.74	8.87	274.35
One-Batch	1.04	1.67	4.11

(a) Results for parameters values from Table 5.1.1 – 1,229,984 experiments.

Algorithm	normalized makespan	rank	degradation from best
UMR	1.00	0.58	0.29
MI-1	1.16	5.76	15.95
MI-2	1.06	2.25	6.58
MI-3	1.09	2.42	9.61
MI-4	1.14	3.53	14.62
MI-5	1.20	4.72	20.08
MI-6	1.25	5.87	25.70
MI-7	1.31	7.00	31.38
MI-8	1.37	8.11	37.09
One-Batch	1.13	4.75	13.52

(b) Results for parameters values in Table 5.1.1 with $cLat < 0.1$, $nLat < 0.1$ – 144,704 experiments.

Table 2. Aggregate comparison of MI-x, One-Batch, and UMR.

forms worse relatively to UMR and MI-x. Indeed, when latencies are small, multi-round algorithm can achieve better overlap of computation and communication by using an increased number of rounds.

5.1.2 Impact of Computation/Communication Ratio on Makespan

In order to provide more insight, we project the results for the parameter space on the B axis. More specifically, for each value of B we compute the makespan of MI-x and One-Batch normalized to that achieved by UMR, Note that the computation/communication ratio of a system is usually key to determining a good schedule. We plot MI-x results only for MI-1, MI-2, MI-3, and MI-4 as trends are identical for $x \geq 4$.

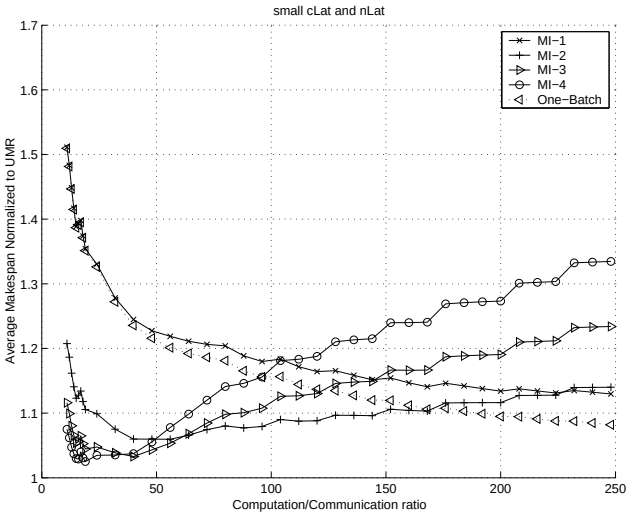

Figure 4. Average makespan of MI-x and One-Batch relative to UMR vs. the computation/communication ratio,$W_{total} = 1000$, $nLat < 0.1$, $cLat < 0.1$.

Small latencies – Figure 4 is for values of $nLat$ and $cLat$ that are lower than 0.1. For larger values we have seen in Table 2(a) that the MI-x algorithm is heavily outperformed by UMR and One-Batch. The One-Batch strategy gets relatively good performance only for large values of the computation/communication ratio. Indeed, overlap of computation and communication is not critical when communication are relatively fast. MI-1 has a similar behavior, but is not as good as One-Batch because it does not take latencies into account. We can see that no MI-x algorithm is effective across the board relatively to UMR.

No latencies – The MI-x algorithm does not take into account latencies. In order to provide a fair comparison with the work in [14], we examined simulation results for $nLat = 0$ and $cLat = 0$ (which we do not believe to be a realistic scenario). In this case, the more rounds the better, and UMR should compute M^* as infinite. Instead, UMR uses an arbitrary upper bound of 50 for M^* (necessary for the bisection solve of Eq. 12). Since we limit MI-x to $x \leq 8$ in our experiments, UMR always leads to the best performance as it always uses 50 rounds. For a fair comparison, we forced UMR to use the same number of rounds as MI-x. As expected, MI-x outperforms UMR because it is not restricted to using uniform rounds and can therefore achieve better overlap between computation and communication. However, UMR is only within 2.1% of MI-x on average. Although this comparison is for an unrealistic platform, it gives us insight into the performance cost of using uniform rounds. Note that when $cLat = 0$ and $nLat = 0$ One-Batch is identical to MI-1.

5.1.3 Summary

The conclusions from our result are:

1. UMR leads to better schedules than MI-x and One-Batch in an overwhelming majority of the cases in our experiments (>95%),

2. Even when UMR is outperformed, it is close to the competing algorithms (on average within 2.04% with a standard deviation of 0.035),

3. Neither MI-x nor One-Batch ever outperform UMR "across the board" (i.e. for a wide range 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 work.

5.2 Impact of Heterogeneity on Makespan

The results we have presented so far have been for homogeneous platforms but the same general trends apply for heterogeneous platforms. Nevertheless, we wish to demonstrate that UMR adequately handles heterogeneous platforms. Therefore we present results for the following experiment. We simulated UMR on a platform consisting of 10 processors with random S_i, $cLat_i$, $nLat_i$ and B_i values sampled from a uniform distributions on the interval $((1 - \frac{het-1}{1+het})mean, (1 + \frac{het-1}{1+het})mean)$, where the means are: $S = 1$, $cLat = 1$, $nlat = 0.1$, $B = 20$. In other words,

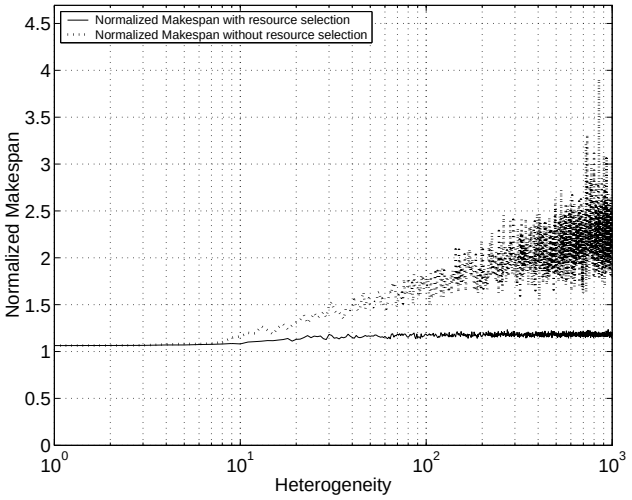

Figure 5. Normalized makespan versus het, with and without resource selection.

processor and link characteristics can differ by as much as a factor het between workers.

Figure 5 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 would 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. 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 1000, UMR still managed to achieve a makespan which is within 20% of the ideal.

In these experiments UMR had to perform resource selection. Indeed, when generating random values for the system parameters, the conditions given in section 4.2 are not satisfied. We must then use fewer resources than available. The key idea is that there may be very slow links connecting the master to very fast processors. For such a processor, the data transfer to that processor during a round completes after other processors have finished computing for the same round, which is detrimental to performance. The resource selection criteria used by UMR is inspired by an approximate version of the constraints given in section 4.2. We sort workers by decreasing values of B_i. We then select the first N' processors out of the original N such that:

$$\sum_{k=1}^{N'} \frac{S_k}{B_k} < 1.$$

To summarize, we give priority to faster links rather than to faster processors, which is very reminiscent of the bandwidth-centric results in [8].

In order to show the benefits of our resource selection method, Figure 5 also plots the normalized makespan versus het when no resource selection is used (dotted curve). One can see that without resource selection, UMR is not able to maintain a low normalized makespan for $het > 20$.

6 Conclusion

In this paper we have presented UMR, an algorithm for minimizing the makespan of divisible workload applications on homogeneous and heterogeneous distributed computing platforms with a master/worker strategy. UMR dispatches work in multiple rounds, which makes it possible to overlap communication and computation. The main question is: How many rounds should be used, and how much work should be sent to each worker during each round? The trade-off is that using many rounds to send small amounts of work to workers allows for good overlapping of communication and computation, but incurs costly overheads. One of our contributions is that we use "uniform rounds": during each round a fixed amount of work is sent to each worker. Although this may appear overly restrictive, it enables us to compute an optimal number of rounds, which was not possible for previously proposed algorithms. We validated our approach via extensive simulations with a realistic platform model and compared it with the multi-round algorithm in [14] and the one-round algorithm in [12]. In our experiments we have seen that UMR leads to better schedules than competing algorithms in the overwhelming majority of the cases (>95%). Neither competing algorithm outperforms UMR "across the board" (i.e. for a large range of computation/communication ratios). Even when UMR is outperformed, it is close to the competing algorithms (within 2.04% on average with a standard deviation of 0.035). Our main result is that 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. We also showed that thanks to its resource selection strategy, UMR can tolerate highly heterogeneous platforms.

In future work we will study the impact of performance prediction errors on the scheduling of divisible workloads. In this paper we have assumed that the scheduler has perfect knowledge of the performance that can be delivered by networks and CPUs and that this performance is constant. In realistic platforms, this assumption does not hold and one must develop scheduling algorithms that can tolerate uncertainties in predicted network transfer times and computation times. The work in [15] addresses uncertainty by reducing the chunk size at each round. The UMR algorithm on the other hand increases chunk size at each round for better performance. We will investigate

an approach that initially increases chunk size for better overlapping of communication and computation, but decreases chunk size towards the end of the application run in order to reduce uncertainties. Our ultimate goal is to implement the resulting scheduling algorithm as part of the APST software [21], an environment for deploying scientific applications on Grid platforms.

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

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

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

[12] 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.

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

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

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