Practical Divisible Load Scheduling on Grid Platforms with APST-DV

Krijn van der Raadt [1] Yang Yang [2] Henri Casanova [1,2]

[1]San Diego Supercomputer Center [2] Dept. of Computer Science and Engineering
University of California, San Diego

Abstract

Divisible load applications consist of a load*, that is input data and associated computation, that can be divided arbitrarily into independent pieces. Such applications arise in many fields and are ideally suited to a master-worker execution, but they pose several scheduling challenges. While the "Divisible Load Scheduling" (DLS) problem has been studied extensively from a theoretical standpoint, in this paper we focus on practical issues: we extend a production Grid application execution environment, APST, to support divisible load applications; we implement previously proposed DLS algorithms as part of APST; we evaluate and compare these algorithms on a real-world two-cluster platform; we show in a case study how a user can easily and effectively run a real-world divisible load application; and we uncover several issues that are critical for using DLS theory in practice. To the best of our knowledge the software resulting from this work, APST-DV, is the first usable and generic tool for deploying divisible load applications on distributed computing platforms.*

1. Introduction

The divisible load application model corresponds to computations that can be arbitrarily divided into *independent* pieces (i.e., they can be executed in any order). This application model is a good approximation of many real-world applications in scientific computing (e.g., see [21, 26, 2, 27, 35, 7, 16, 33, 3, 19, 14]). Divisible load applications are amenable to the simple master-worker programming model and can in principle be easily executed on platforms ranging from one single cluster to large distributed Grids.

This paper is based upon work supported by the National Science Foundation under Grant No. 0234233.

Two key challenges faced by users for executing parallel applications on distributed platforms are *easy deployment* and *high performance*, and divisible load applications are no exceptions. The deployment challenge arises on Grid platforms as they contain heterogeneous resources with various access methods and policies. The current Grid middleware infrastructure [18] provides most of the required functionality for Grid application deployment, but it is complex and not designed to be used directly by end users. A successful approach has been to provide so-called "application-level tools" that isolate the user from the middleware infrastructure and take on the burden of all application deployment logistics [5].

A great amount of research has been done on scheduling Divisible Load applications (see Section 2.2), but most of this work, including our own, has been either purely theoretical or specific to a single application. So the goal of this paper is *not* to bring up new algorithms; instead, our **first contribution** in this paper is to provide a generic application-level tool for the easy deployment of these applications on Grid platforms. We build upon an existing Grid application-level tool, APST [11, 4], to which we add support for divisible load applications. With this new tool, which we call APST-DV, users can deploy applications on a wide variety of resources completely automatically and transparently. We use a case study to demonstrate how users can do this easily and effectively. Our **second contribution** is a practical evaluation and comparison of these algorithms, obtained by running APST-DV on a real-world platform. Beyond demonstrating APST-DV's functionality, these experiments allow us to identify issues relevant to the use of divisible load theory in practice.

This paper is organized as follows. We present background on divisible load in Section 2. Section 3 briefly describes the APST software and highlight the key aspects of our implementation of APST-DV, including the scheduling algorithms. We present our experimental results in Section 4, followed by our case study in Section 5. Section 6 concludes the paper with future directions.

2. Divisible Load: Applications and Scheduling Algorithms

In this section we define the divisible load model, give examples of real-world divisible load applications, and provide a small survey that highlights the spectrum of application characteristics. We then review relevant previous work in the area of divisible load scheduling.

2.1. Divisible Load Applications

The input to a divisible load application consists of many *small independent* parts, and the processing time of each part is small compared to the time to process the whole input. So the total input can be *divided* into *chunks* of *arbitrary sizes*, which may be processed in any order (and each chunk may itself contain an arbitrary number of small parts). Correspondingly, the computation can be easily decomposed into sub-tasks and can thus be easily deployed on distributed computing platforms in a master-worker fashion. Many real-world applications fit the *divisible load* model. Divisible load applications are similar to many so-called "embarrassingly parallel applications", but the term "divisible load" is used to emphasize that these applications are data-intensive and that communication takes a non-negligible amount of time. In fact, a large part of the difficulty of achieving high performance for these applications comes from the need to orchestrate communication and computation.

To get an idea of the large diversity of the characteristics of divisible applications, we conducted simple experiments for three specific applications: (1) HMMER [21], a bioinformatics sequence comparison application; (2) MPEG4 video compression [26]; and (3) VFleet [32], a volume rendering application.

Table 1 shows for each of these three applications the input size in MB, the running time in seconds on an Athlon 1.8GHz, and the computation-communication ratio, r, computed assuming a 100 Mb/s data transfer rate. The table also shows data for the Data Mining application presented in [30]. The main point to draw from the data is that these applications exhibit different characteristics, and in particular a wide range of values for r (differences of more than one order of magnitude).

The fifth column in Table 1 shows the coefficient of variance (i.e., standard deviation divided by the mean, in percentage) of the amount of computation per unit of load, which we call γ. We can see that some applications exhibit a γ value up to approximately 10%, due to data-dependent and/or non-deterministic computation. In terms of scheduling, this implies that there will be some uncertainty when predicting the computation time of a chunk of load, which can negatively impact the schedule.

2.2. Divisible Load Scheduling

An important problem whose solution holds the key to high performance for divisible load applications on distributed computing platforms is Divisible Load Scheduling (DLS): the decision process by which the load is divided and assigned to compute resources, with the goal of minimizing application "makespan", i.e. execution time.

The first proposed DLS algorithms were *One-Round* algorithms, so called because they assign exactly one chunk of the load to each worker. These algorithms were studied for many platform topologies (e.g., Linear Networks, Single-level Trees, Meshes, Hypercubes) and we refer the reader to [6] for references to specific papers. Most of these algorithms assume purely linear cost for transfer and computation, that is the time to transfer some amount of data is proportional to the data size. The most recent ones consider communication start-up costs, i.e. they assume an *affine* communication cost model, which is known to be more realistic as real networks do experience start-up costs (e.g., latencies, overhead for establishing connections).

One clear limitation of One-Round algorithms is that they do not overlap communication with computation well, which has led to the development of *Multi-Round* algorithms that assign multiple chunks to each worker in rounds and increase chunk size throughout application execution in an attempt to pipeline communication and computation. Much fewer results are available for Multi-Round algorithms than for One-Round algorithms and they are all on single-level tree topologies. The work in [8] proposes a multi-round algorithm that assumes purely linear communication and computation costs. [37] extends this algorithm to affine costs for both communications and computations, which is more representative of real-world platforms. Both these algorithms assume that the number of rounds is magically fixed and are only applicable to homogeneous platforms. By contrast, the UMR algorithm in [39] computes a near-optimal number of rounds with affine communication and computation costs, and it is applicable to heterogeneous platforms.

Finally, the recently proposed RUMR algorithm [38] extends UMR and attempts to mitigate the effects of *uncertainty* on chunk communication and computation times. The RUMR approach is to first increase chunk size for better pipelining, as UMR, but decrease chunk size towards the end of the application execution to tolerate uncertainty. The notion of decreasing chunk size for better robustness to uncertainty was pioneered by the GSS and Factoring approach [22, 20, 23].

In this work we focus on Multi-Round algorithms. We target distributed Grid platforms that aggregate multiple parallel computing platforms, typically commodity clusters. These platforms can be easily modeled as single-level trees

Application	input size (MB)	running time (sec)	r	γ	$\frac{\max-\min}{\text{mean}}$
HMMER	802.0	534	6.7	9%	2700%
MPEG	716.8	2494	34.8	10%	30%
VFleet	87.5	600	68.0	1%	2%
Data Mining	400.0	3150	78.0	N/A	N/A

Table 1. Characteristics of 4 divisible load applications: input data size, running time on a 1.8GHz Athlon, communication/computation ratio (r) assuming a 100Mb/sec network, coefficient of variation of the running time of a unit of load (γ), and percentage spread of the running time of a unit of load ($\frac{max-min}{mean}$).

in which each leaf is a cluster and the root is the master holding the application's input data, which makes Multi-Round algorithms applicable. We refer the reader to recent surveys [9, 28], to the special issue of the Cluster Computing journal [1], and to the Web page collecting related literature [29] for more details about DLS research.

3. The APST-DV Software

3.1. APST Background

APST [4, 12] is a Grid application execution environment originally targeted to "Parameter Sweep Applications" that consist of a fixed number of independent tasks. The APST software was designed with the goal of fully automated and transparent deployment of applications on Grid platforms, as well as high performance via efficient scheduling. APST runs as two distinct processes: a daemon and a client. The daemon is in charge of deploying and monitoring applications. Its central component is a scheduler that makes all resource allocation decisions. The client is essentially a console (several APIs are also available) that can be used by the user to interact with the daemon (e.g., to submit requests for computation). The user interface is XML-based and typically no modification of the application is required.

APST is currently used in production for a number of applications, including the MCell neuroscience application [10], the Encyclopedia of Life (EOL) bioinformatics application [25], the Vizport visualization portal [34], and the discrete-event simulation application SIMGRID [24]. We refer the reader to [12, 4] for more details about APST.

3.2. APST-DV: Motivation and Design

APST is not well-suited to divisible load applications as it expects a finite and complete list of application tasks as input. As a result, current divisible load application users are forced to divide the load manually into some number of sub-tasks. However, the field of DLS research shows that load division is a difficult problem and that simple solutions (e.g., divide the load in many identical pieces) are bound to achieve poor performance. So while APST does the best it can with the divided load submitted by the user, different division schemes that account for both application and resource characteristics would inherently allow higher performance.

The popularity of APST is mostly due to the fact that it does not require modification of the application, requires only a minimal understanding of XML, and can be used immediately within a small local-area network with default mechanisms. Users can then easily and progressively transition to larger scale Grids because APST transparently builds on the base Grid software infrastructure. We wish to build on these strengths and extend APST to support divisible load applications. We call this extension APST-DV.

APST-DV needs to accomplish the following. It must provide a way for the user to specify a divisible load application in XML. It needs to divide the load into individual tasks (or "chunks"). This must be done according to a DLS algorithm. Such algorithms typically require information about the application and the resources (e.g., how fast one unit of load runs on a given resource), and APST-DV must obtain such information automatically. The chunks must then be sent out to storage resources and computation must be initiated on remote compute resources, which can be accomplished easily as APST already provides mechanisms for accessing a wide range of resources. Finally, output from chunk computation needs to be returned to the users and, most likely, "glued" together. This last step is typically application-specific and we leave it to the user. We briefly review interesting aspects of our implementation of APST-DV below.

3.3. XML Divisible Load Specification

We have added a new XML element to APST, `divisibility`, within the existing `task` construct. See Figure 1 for a sample divisible load specification, and the APST webpage [4] for a complete

```
<task
  executable="a_divisible_app"
  input="bigfile"
>
  <divisibility
  input="bigfile"
  method="uniform"
  start="0"
  steptype="bytes"
  stepsize="10"
  algorithm="rumr"
  probe="probefile"
  />
</task>
```

Figure 1. Sample APST-DV XML specification of a divisible load application.

description of APST's XML schema. The `input` attribute specifies the file(s) that contain the load's input data that must be divided. The `method` attribute specifies the method used for dividing the input file(s), which will be described in more detail in Section 3.4. In this example the method used is `uniform`, which is defined by the following three attributes: the `start` attribute specifies the starting offset in the load, that is not to be used in a chunk of load; the `steptype` attribute specifies the type of load unit, for example *bytes*; and the `stepsize` attribute is used to indicate how many load units can go in a chunk. In this example the input file can be divided at each 10-byte boundary starting at byte 0 (meaning that the size of each load chunk in bytes will be a multiple of 10). Note that APST-DV divides the load on-the-fly, thereby avoiding creating a prohibitive number of files for each individual chunk.

In our current prototype the `algorithm` attribute specifies which DLS algorithm to use for scheduling the applications (`rumr` in the example). Eventually this could be determined automatically by APST. The meaning of the `probe` attribute will be explained in Section 3.5.

3.4. Load Division Methods

In the ideal divisible load model the input can be divided continuously, exactly as the scheduling algorithm dictates. However, depending on the application, some cut-off points in a load would be valid, and some would not. To enable the user to specify where the load can be divided, APST-DV implements three methods to determine what the closest valid cut-off point is to the cut-off point that is requested by the scheduling algorithm.

Uniform – With the uniform division method a cut-off point can be at some number of load units from the beginning of the load. There are two types of load unit, specified by the `steptype` attribute, which are `bytes` and `separator`. A valid cut-off point is measured in the number of *steps* from the beginning of the load, where a step is a number of load units. This step size is specified by the `stepsize` attribute (see Figure 1). If the load unit is `bytes`, and the step size is 10, it means that the load can be cut at every 10 bytes from the start. For the `separator` type a valid cut-off point is indicated by the occurrence of a special separator character or characters in the load. This separator character is specified in the APST-DV XML specification by the `separator` attribute.

Index – With the index division method, the user supplies an index file, containing an entry for every valid cut-off point. For every desired cut-off point the scheduling algorithm consults the index file to find the nearest valid cut-off point. Cut-off points are specified as numbers of bytes from the beginning of the load file. The index file is specified in the XML specification by the `indexfile` attribute.

Callback – APST-DV also provides the callback division method, which allows the user to supply a program to perform load division. APST-DV can call this program, passing it an offset and a chunk size in the `load` attribute, both specified in terms of number of "work units" whose sizes are application-specific. The callback programs then extracts the chunk data from the load and places it into a temporary file that APST-DV can send to remote resources to initiate chunk computation. The callback program and its possible command line arguments (for instance a typical callback program would take the name of the input file containing the entire load as a command-line argument) are specified in the XML specification by the `callback` and the `arguments` attributes.

3.5. Collection of Resource Information

DLS algorithms, like most scheduling algorithms, make their decisions based on application and resource information. There are two approaches to gather such information. The first approach is to rely on application performance models and on resource information provided by services such as MDS [13], NWS [36], and Ganglia [15]. Some of this information can be dynamic and must be retrieved periodically. The advantage of this approach is that it is lightweight. The drawback is that it requires an infrastructure to be installed, and that it is often difficult in practice to obtain accurate estimates of computation and transfer times for a particular application based on monitored resource information. The second approach is to just observe application performance for a few application tasks and data transfers, and use this observation to estimate the performance of all application components. This approach is more costly

as real work needs to be done to obtain performance information, although this work can be useful to the application, but more accurate as the performance delivered by the resources is experienced directly at the application level.

Since our target applications typically exhibit long execution times, we opted for the second approach. This approach has actually been explored in the context of DLS in [17]. The idea is to "probe" the resources by sending out a relatively small chunk of the overall load to each available resource and observing chunk transfer time and chunk execution time. We use a very simple probing strategy in our current APST-DV implementation: we do a round of probing, and then start the real application execution. The load we use for probing is not part of the actual load, but instead consists of a separate, user-specified small input file that is representative of the application's load. "Representative" may mean "close to the average case" for scenarios in which there is uncertainty on the computational cost of a unit of load (see Section 2.1). In such scenarios, the scheduling will be impacting by performance prediction errors. The input file used for probing is specified by the `probefile` attribute in the XML specification of a divisible load application.

Finally, some of the scheduling algorithms implemented in APST-DV require estimates for communication and computation start-up costs. APST-DV obtains these estimates periodically by launching no-op jobs on each worker and transferring empty files to storage resources.

3.6. Scheduling in APST-DV

The current APST-DV prototype implements the following four DLS scheduling algorithms:

SIMPLE-n – uniformly divides the input among the workers, and divides the data for each worker into n chunks. No probing is used. This is the simplistic "static chunking" approach that is currently used by divisible load application users who use APST. We used SIMPLE-1 and SIMPLE-5 in our experiments.

Uniform Multi-Round (UMR) [39] – a recently proposed DLS algorithm that (i) is designed to maximize communication/computation overlap; (ii) uses multiple rounds; (iii) accounts for communication and computation start-up costs; (iv) computes a near-optimal number of rounds; and (v) can be used on heterogeneous platforms. Points (iii)-(v) above represent significant advances over previously proposed algorithms and make multi-round DLS feasible in practice. (See Section 2.2 for a brief discussion of multi-round DLS.) UMR increases chunk size geometrically throughout execution to achieve good pipelining of communication and computation. This algorithm uses probing.

Weighted Factoring [23] – divides the load into chunks in rounds, and decreases chunk size by 2 between rounds (down to a minimal chunk size). Chunks are sent out to workers in a greedy fashion. The algorithm is called "weighted" because the size of a chunk assigned to a worker is proportional to the worker's speed, which is known to achieve better load-balancing than plain factoring. Our implementation of weighted factoring uses probing. It also observes chunk execution times throughout application execution to refine its estimates of worker speeds. SIMPLE-n and UMR do not perform such adaptation. The factoring method was specifically designed to deal with uncertainty in computation times: application execution ends with small chunks, which makes it easier to do load-balancing. However, Factoring was not designed to maximize overlap of communication and computation.

Robust Uniform Multi-Round (RUMR) [38] – One problem with UMR is that, unlike Factoring, it was not designed to tolerate uncertainty on chunk transfer/execution times (execution ends with large chunks). To achieve the best of both worlds, the RUMR algorithm splits application execution into 2 phases. During the first phase chunk size is increased using the UMR algorithm, and during the second phase chunk size is decreased using Weighted Factoring. The RUMR algorithm uses a heuristic to determine when to start the second phase. We also experiment with a version of RUMR called **Fixed-RUMR** presented in [38] that always schedules 80% of the load in the first phase. RUMR uses probing.

Some of the above algorithms have been evaluated in simulation in previous work. For instance, in [39] it was shown that UMR outperforms competing multi-round algorithms and largely outperforms SIMPLE-n. In [38] it was shown that RUMR outperforms both UMR and Factoring for a wide range of uncertainty on chunk compute and transfer time. While these results are valuable, our goal here is to run these algorithms in the real world and observe what truly happens. In fact, just going through the process of implementing these algorithms as part of usable software has highlighted several interesting practical issues.

4. Experimental Evaluation

4.1. Methodology

Application – We have seen in Section 2.1 that the fundamental characteristics of divisible load applications span a range of values. Rather than picking one single application, which would limit the space of our evaluation, or trying to run a large number of different applications, which would require a lot of unnecessary effort, we opted for using a synthetic application. Note that we have tested APST-DV with the real-world applications mentioned in Section 2.1, and that we present a case study with a real application in Sec-

tion 5. Our synthetic application reads in an input file and does some floating point operations in a loop. This synthetic application can be tuned to exhibit specific application characteristics: in particular, the communication/computation ratio, r, and the uncertainty on load unit computation time, γ (we use a Normal distribution for generating random computational costs for units of load).

Computing Platform – We used a small Grid consisting of two clusters: the Meteor cluster at the San Diego Supercomputer Center (SDSC), which consists of 57 dual-processor Pentium III 790∼996MHz nodes; and the DAS-2 cluster at Vrije Universiteit in Amsterdam, the Netherlands, which consists of 72 dual-processor 1Ghz Pentium-III nodes. We access the clusters via the SGE and PBS batch schedulers. The APST-DV daemon and initial input for the divisible load application were located in the Grid Research and Innovation Laboratory (GRAIL) at UCSD, about 1/2 mile from SDSC. In this section, we focus on platforms whose processors are dedicated during application execution. This is so that we can control the performance prediction error parameter γ in our experiments.

Uncertainty – We wish to study the effect of uncertainty, which causes performance prediction errors, on divisible load scheduling. Indeed, some of the DLS algorithms described in Section 3.6, namely RUMR and Factoring, have been specifically designed to tolerate performance prediction errors, and we wish to evaluate how robust they are in practice. Uncertainty can come from two sources: the application itself, and the compute platform. As seen above, we experiment with $\gamma = 0\%$ and $\gamma = 10\%$, with the latter generating inherent uncertainty in the chunk execution time. With a dedicated computing platform and stable network, the only significant source of uncertainty in our setup is the application itself which allows us to control our experiments.

4.2. Experimental Results

We ran APST-DV with all the DLS algorithms described in Section 3.6, back-to-back. Each data point corresponds to an average over 10 distinct runs. Each application run lasted between 68 minutes and 178 minutes, depending on the resources and the scheduling algorithm used.

DAS-2, 16 nodes, r = 37, γ = 0, 10 – We first ran our application on the DAS-2 cluster only. For each algorithm we compute the (average) application makespan achieved. Results are shown in Figure 2 for $\gamma = 0$ and $\gamma = 10$.

For $\gamma = 0$ we found expected results. The RUMR and UMR algorithms (note that in this case we have no uncertainty and RUMR degenerates to pure UMR) lead to the best performance as they overlap communication and computation well and account for the large start-up costs for communication and computation (around 6.4s and 0.7s respectively in this case). The second closest algorithm is SIMPLE-5 (5% slower), while SIMPLE-1 is 26% slower. The Factoring algorithms are roughly 10% slower than UMR/RUMR, due to poor overlap of communication with computation. These results confirm the simulation results presented in [39, 38]. One may ask why communication is an issue, since the r value of 37 seems to mean that communication time would be only $1/37$ of the total makespan. One should note that communications to workers are serialized. So when multiple workers are used, the communication time does not decrease, while the communication decreases. As a result, communication represent a more significant part of the makespan as the number of worker increases.

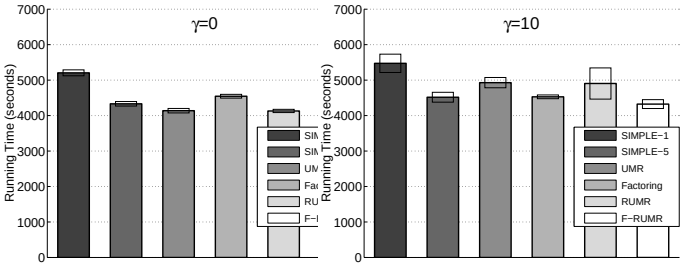

Figure 2. DAS-2, 16 nodes

For $\gamma = 10$, that is with more uncertainty, one expects Weighted Factoring to perform better than UMR, which is the case (e.g., Weighted Factoring is about 8% faster than UMR). The simulation results in [38] indicate that RUMR should outperform Weighted Factoring as it strives to both overlap communication and computation, and to mitigate the effects of uncertainty. However, in our experiments, RUMR exhibits poor performance when compared to Weighted Factoring. After looking into the detailed execution report generated by APST-DV, this is what we found: the RUMR algorithm as developed in [38] assumes that the value for γ is known in advance and, using this value, pre-determines when the second phase (i.e., the Factoring phase) should begin. However, in our experiments, the value of γ is "discovered" throughout application execution. We found that in most cases, when RUMR discovers that it should switch to its Factoring phase, it is too late and the last round (which is large since UMR increases chunk size) has already been started. This prevents RUMR from doing a late switch to its second phase, meaning that Factoring is in fact never used. This is a good example of an aspect of DLS theoretical research that does not translate well to practice. This observation highlights a major limitation of the RUMR algorithm (although it may be argued that the magnitude of the uncertainty could be learned from past application exe-

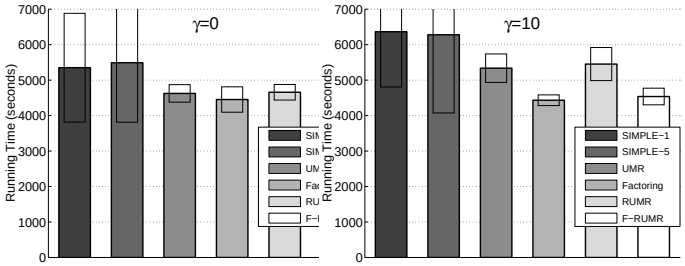

Figure 3. Meteor, 16 nodes

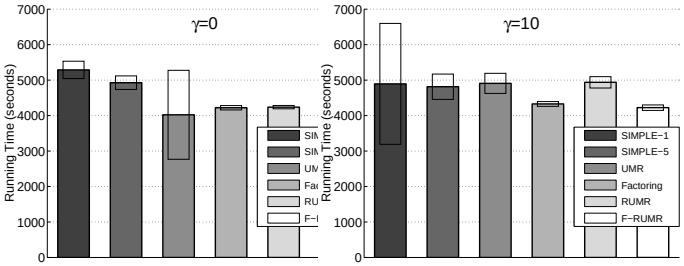

Figure 4. Meteor+DAS-2, 16 nodes

cutions). Importantly, we can see that the Fixed-RUMR algorithm does the best in our experiments, therefore justifying that RUMR's two-phase approach is sound, provided there is a mechanism for switching to the second phase in time.

Meteor, 16 nodes, r = 46, γ = 0, 10 – Using only the Meteor cluster we have a higher value for r and obtained the results shown in Figure 3.

For $\gamma = 0$ we can see that all algorithms achieve comparable performance, except for SIMPLE-1 and SIMPLE-5, which are 21% and 24% slower than the best algorithm. In this environment start-up costs are low (around 0.7s for communication and 0.1s for computation) since the Meteor cluster is close to the APST daemon. (The network bandwidth is also marginally higher: around 116 kB/sec compared to 92 kB/sec to the DAS-2 cluster.) As a result, the UMR approach does not lead to any advantage as it is really designed to handle situations in which start-up costs are significant.

For $\gamma = 10$, the only thing that matters for performance in this environment is adaptation to uncertainty and clearly the Weighted Factoring approach is the best. UMR and RUMR (20% and 23% slower) suffer from the same problems as discussed above for the DAS-2 experiments. But, importantly, Fixed-RUMR leads to roughly the same performance as Weighted Factoring.

These results show that if the platform is a nearby dedicated cluster, then a simple Factoring approach is sufficient, which is not surprising.

DAS-2 (8 nodes) + Meteor (8 nodes), γ = 0, 10 – In these experiments we used nodes from the two clusters, so the communication/computation ratio was a mix of the ones for the two previous experiments. Results are shown in Figure 4. The results here show that with no uncertainty ($\gamma = 0$), UMR and RUMR lead to the best performance (again, they are identical in this case) and that SIMPLE-1 and SIMPLE-5 have poor performance (25% and 17% slower). When there is uncertainty ($\gamma = 10$), Weighted Factoring and Fixed-RUMR lead to the best performance. Once again, the SIMPLE-1 and SIMPLE-5 algorithms do not perform well (28% and 14% slower).

4.3. Discussion

From the experimental results above (we also ran experiments with different subsets of our clusters and different load sizes, but did not learn anything different) we draw the following broad conclusions:

1. The SIMPLE-n algorithm, which is what current APST users are using for running divisible load applications, is always inefficient (on average SIMPLE-1 and SIMPLE-5 are 28% and 18% slower than the best algorithm). As a result, our work on APST-DV has already significantly improved the state of practical deployment for these applications.
2. The UMR approach is best when uncertainty is low, as it accounts for communication and computation start-up costs, and overlaps communication with computation well. Its performance is poor when uncertainty becomes significant (on average 17% slower than the best algorithm).
3. Expectedly, when the platform consist of a single, nearby cluster, then a simple Factoring approach is sufficient.
4. The general RUMR approach is the most effective across the board for low and high uncertainty, but the algorithm as it was proposed in [38] does not do well in practice. Indeed, it does not switch to its second phase in time. This was shown by the good performance exhibited by the Fixed-RUMR version. A key direction for further RUMR development is to solve this problem and in the meantime Fixed-RUMR can be used by APST-DV users.

5. Case Study: MPEG-4 Encoding

In this case study we use APST-DV to run a *parallel MPEG-4 encoding* application to compress a DV format video file, shot with a digital video camera. There are many MPEG-4 encoders available and we use *mencoder* [26], which is an open source command line tool for decoding, filtering, and encoding video and audio files. One of the

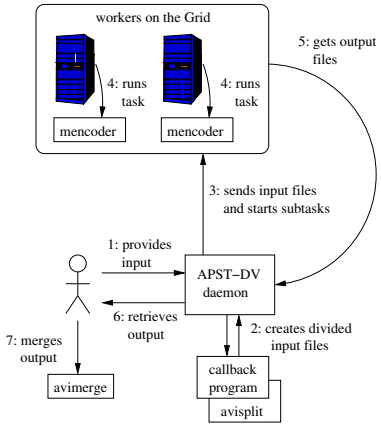

Figure 5. Case study scenario.

```
<task
  executable="run_mencoder.sh"
  arguments="input.avi mpeg4.avi"
  input="input.avi"
  output="mpeg4.avi"
>
 <divisibility
   input="input.avi"
   method="callback"
   load="1830"
   callback="callback_avisplit.pl"
   arguments="input.avi"
   algorithm="rumr"
   probe="probe.avi"
   probe_load="21"
 />
</task>
```

Figure 6. APST-DV XML application specification used in the case study.

merits of APST-DV is that it can deploy readily available applications, such as mencoder, without the need to modify them.

5.1. APST-DV Usage

APST-DV needs to be able to divide the input file's load into chunks, which in the case study is done using the *callback* division method. As seen in Section 3.4 this method uses an external program to create the load chunks. It is up to the end user to provide this callback program, but APST-DV provides two example callback programs (written in C and Perl) that are easily modified and adapted to a particular application. In our setup we modified the Perl example callback program. We managed to keep the modifications to the Perl script very simple, because we use a readily available tool called *avisplit* to do the actual work. This tool can be used to divide any AVI file into smaller files, and the Perl script is just a wrapper around the avisplit program. Another tool named *avimerge* can be used by the end user to merge all the output files together. Both these tools are part of the open source *transcode* [31] video stream processing toolkit.

In Figure 5 we can see step by step how the divisible load application is run:

1. The user provides the APST-DV daemon with the input file(s) and the XML specification of the divisible load application.
2. The APST-DV daemon uses the callback program, which uses the avisplit tool, to create the load chunks.
3. The APST-DV daemon sends the chunks to the workers on the Grid, and starts an encoding task on each worker.
4. The workers use the mencoder program to encode their chunk of load.
5. The APST-DV daemon downloads the output files from the workers.
6. The end user retrieves the output files from the APST-DV daemon.
7. The end user uses the avimerge tool to merge the output files together into one output file.

This shows how APST-DV makes it straightforward for the end user to run a real-world divisible load application. All the end user has to do to setup the system is create the XML specification file and modify the example callback program to create a wrapper around an existing tool. We describe the XML specification in what follows.

As far as APST-DV is concerned, the application consists of a specification of input and output files, an executable with command-line arguments, and a specification of the application's divisibility. Figure 6 shows the `task` part of the XML specification file that is used in the case study. Both the input file (`input.avi`) and the probe file (`probe.avi`) are DV encoded movies, and the output file (`mpeg4.avi`) is an MPEG-4 encoded movie. The `input.avi` file is 209 MB in size, and contains 1,830 frames, which comes down to about 1 minute of video footage. The `probe.avi` file is 2.4 MB in size, and contains 21 frames, which is 0.7 seconds of video footage.

The load division method that is used is the *callback* method, and the callback program that is used is a Perl script called `callback_avisplit.pl`. As described earlier, this script uses the *avisplit* tool to create chunks by specifying the desired range of frames. Note that the load in this case study is measured in *frames* instead of *bytes*. This alternative load size is specified in the XML specification, shown in Figure 6, using the `load` and `probe_load` attributes. The number of frames in the divisible input file that we use (`input.avi`) is 1830, and the probe file (`probe.avi`) contains 21 frames. Note that a movie file containing an hour of footage has a load of 108,000 frames

and is about 12 GB in size. The callback division method enables us to use the avisplit tool, rather than developing our own tool. This demonstrates that APST-DV provides a flexible way for users to define their loads, and integrates well with tools that may be at the user's disposal.

5.2. Experimental Runs

For this case study we used a platform consisting of 6 hosts at the Grid Research and Innovation Laboratory (GRAIL) at UCSD. It is a collection of non-dedicated Linux workstations on a single 100Mb/sec LAN. Because one of the hosts has two CPUs, this comes to a total of 7 processors, with speeds of 700MHz (1 x AMD Athlon) and 1.73 GHz (6 x AMD Athlon XP). In this run all compute resources are accessed via Ssh and files are moved using Scp. This configuration is easily described to APST-DV using the traditional APST XML resource description schema [4].

We ran 10 runs of the application for each scheduling algorithm, and each run lasted between 12 and 21 minutes. As the hosts that were used for this experiment were not dedicated to our application, there is some uncontrolled uncertainty in chunk execution time, and the results comparison of the different scheduling algorithms are not as dependable as the results of the experiments described in Section 4. The average value for γ (the coefficient of variance of the amount of computation per unit of load) that was measured in this experiment is 20%, and the communication/computation ratio r is 13.5.

Because of the fluctuation of resource availability, it was to be expected that the more adaptive algorithms would work best in this environment. Furthermore, due to the fast network, overlap of communication with computation is not critical. And Indeed, Weighted Factoring leads to the best performance. Interestingly, RUMR's performance is roughly the same (within 2%). By contrast to the other experiments that we ran, the RUMR algorithm successfully switches to its second phase in every one of the ten runs, due to the fact that the γ value was higher than in the experiments in the previous section. This may indicate that the phase switching problem identified in the previous section is only faced for moderate uncertainty, which we will confirm with further experimentation. UMR and Fixed-RUMR perform very similarly, around 7% slower than Weighted Factoring, as they do not account for uncertainty sufficiently. As expected, Simple-5 and Simple-1 do not perform well, 38% and 52% slower than Weighted Factoring.

6. Conclusion

In this paper we have presented and evaluated APST-DV, an extension to the APST Grid application-level tool to support *Divisible Load Applications*. To the best of our knowledge, our work provides the first generic software environment to deploy them on current distributed computing platforms. We have demonstrated the use of APST-DV with an MPEG-4 encoding case study, showing that a user can easily and effectively use readily available tools together with APST-DV to run real-world divisible load applications. APST-DV embeds a scheduler that currently implements four Divisible Load Scheduling algorithms. We experimentally evaluated these algorithms on a real-world testbed consisting of two geographically distant clusters. Our experiments show that the simplistic "static chunking" approach used by current APST users to run divisible load applications is not effective. Among other results, we have found that the RUMR approach proposed in [38] is the most effective across the board. One limitation of this approach, however, is that a better mechanism for switching between the two phases of its execution is needed. For now we have addressed this problem with a simple version of the algorithm, Fixed-RUMR, that performs well in practice.

In future work we will investigate new ways in which RUMR can switch to its second phase appropriately. We will also implement an adaptive version of RUMR that updates its view of the platform after each sub-task completes. The results in this paper validate our prototype implementation of APST-DV, and we will release the software as part of the APST v2.3 distribution.

Acknowledgments

We wish to thank the San Diego Supercomputer Center and the Vrije Universiteit in Amsterdam for allowing us to use their compute resources. We are also grateful to James Hayes for his help with APST.

References

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

[2] 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.

[3] N. Amano, J. o Gama, and F. 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.

[4] The APST Project. `http://grail.sdsc.edu/projects/apst`.

[5] H. Bal, H. Casanova, J. Dongarra, and S. Matsuoka. Application-Level Tools. In *Grid 2: Blueprint for a New Computing Infrastructure*. John Wiley, second edition, 2003. Foster, I. and Kesselman, C., editors.

[6] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree net-

works: results and open problems. Technical Report RR-2003-41, LIP, École Normale Supérieure de Lyon, September 2003.

[7] W. Bethel, B. Tierney, J. lee, D. Gunter, and S. Lau. Using high-speed WANs and network data caches to enable remote and distributed visualization. In *Proceedings of Supercomputing (SC'00)*, 2000.

[8] 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.

[9] V. Bharadwaj, D. Ghose, and T. Robertazzi. A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–18, 2003.

[10] H. Casanova, T. Bartol, J. Stiles, and F. Berman. Distributing MCell Simulations on the Grid. *International Journal of High Performance Computing Applications*, 14(3):243–257, 2001.

[11] 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.

[12] H. Casanova and F. Berman. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 33. John Wiley & Sons Publisher, Inc., 2003.

[13] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE Symposium on High-Performance Distributed Computing (HPDC-10)*, August 2001.

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

[15] The Ganglia Project. http://ganglia.sourceforge.net.

[16] A. Garcia and H.-W. 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.

[17] D. Ghose, H. J. Kim, and T. H. Kim. Adaptive Divisible Load Scheduling Strategies for Workstation Clusters with Unknown Network Resources . Technical Report KNU/CI/MSL/001/2003, Department of Control and Instrumentation Engineering, Kangwon National University, Korea.

[18] The Globus Project. `http://www.globus.org`.

[19] S. Goil and A. 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.

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

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

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

[23] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load Sharing in Heterogeneous Systems via Weighted Factoring. In *Proceedings from 8'th Symposium on Parallel Algorithms and Architectures*, pages 318–328, 1996.

[24] 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.

[25] W. Li, R. Byrnes, J. Hayes, V. Reyes, A. Birnbaum, A. Shabab, C. Mosley, D. Pekurowsky, G. Quinn, I. Shindyalov, H. Casanova, L. Ang, F. Berman, M. Miller, and P. Bourne. The Encyclopedia of Life Project: Grid Software and Deployment. *Journal of New Generation Computing on Grid Systems for Life Sciences*, 2004. to appear.

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

[27] G. Miller, D. G. Payne, T. N. Phung, H. Siegel, and R. Williams. Parallel Processing of Spaceborne Imaging Radar Data. In *Proceedings from Supercomputing (SC'95)*, 1995.

[28] T. Robertazzi. Ten reasons to use divisible load theory. *IEEE Computer*, 36(5):63–68, 2003.

[29] T. G. Robertazzi. Divisible Load Scheduling. `http://www.ece.sunysb.edu/~tom/dlt.html`.

[30] T. Tamura, M. Oguchi, and M. Kitsuregawa. Parallel database processing on a 100 Node PC cluster: cases for decision support query processing and data mining. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–16, November 1997.

[31] The transcode project. `http://www.theorie.physik.uni-goettingen.de/~ostreich/transcode`.

[32] Vfleet volume rendering package. `http://www.psc.edu/Packages/VFleet_Home`.

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

[34] The Scalable Vizualiation Toolkit VizPortal project. `https://gpadev.sdsc.edu/dev/swhitmor/visPortal/`.

[35] A. Watt. *3D Computer Graphics*, chapter 13. Addison-Wesley.

[36] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999.

[37] 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.

[38] 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)*, June 2003.

[39] 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.