Title
APST-DV: Divisible Load Scheduling and Deployment on the Grid

Authors
van der Raadt, Krijn
Yang, Yang
Casanova, Henri

Publication Date
2004-04-28

Peer reviewed

APST-DV: Divisible Load Scheduling and Deployment on the Grid

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 have received a lot of attention in recent scheduling literature. These applications consist of an amount of computation, or *load*, that can be divided arbitrarily into independent pieces. The problem of Divisible Load Scheduling (DLS) has been studied extensively, but mostly from the theoretical standpoint. In this paper we focus on practical issues and make the following contributions: we implement previously proposed DLS algorithms as part of a generic production Grid application execution environment, APST; we evaluate and compare these algorithms on a real-world two-cluster platform; and we uncover several issues that are critical for using DLS theory in practice effectively. 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 current distributed computing platforms.

1 Introduction

The divisible load application model corresponds to computations that can be divided arbitrarily into arbitrary numbers of *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 [25, 33, 2, 34, 45, 15, 9, 20, 42, 15, 4, 23, 17]. Divisible load applications are amenable to the simple master-worker programming model and can therefore be easily implemented and deployed on current parallel computing platforms ranging from one single cluster to several clusters aggregated in a Grid platform.

For most parallel applications that must be executed on a distributed platform, two key challenges for users are *easy deployment* and *high performance*, and divisible load applications are no exceptions. The deployment challenge is particularly severe on Grid platforms as they contain heterogeneous resources with a range of access methods and policies. Furthermore, while the current Grid middleware infrastructure [22] provides most of the required functionality for Grid application deployment, it is complex and not designed to be used by end users (i.e., disciplinary scientists). A successful approach to solve this challenge has been to provide so-called "application-level tools" that take on the burden of all application deployment logistics on behalf of the user. A taxonomy of such tools is available in [6]. Previous work on divisible load applications, including our own, has been either purely theoretical or specific to a single application. 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 on an existing Grid application-level tool, APST [13, 5], which is currently used in production for several Grid applications [29, 32, 43, 28], and to which we add support for divisible load applications. With this new tool, which we call APST-DV, users can deploy their applications on a wide variety of resources completely automatically and transparently.

The challenge of divisible load application performance has received a lot of attention in the literature and many authors have proposed *divisible load scheduling* (DLS) algorithms. Consequently,

we have implemented some of the most recent and efficient DLS algorithms within APST-DV. Our **second contribution** in this paper 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 applications and divisible load scheduling in Section 2. Section 3 briefly describes the APST software and highlight the key aspects of our implementation of APST-DV, including the DLS algorithms. We present our experimental results in Section 4, and Section 5 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. Then we review relevant previous work in the area of DLS.

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 total 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 application can be easily decomposed into sub-tasks and can thus be easily deployed on distributed computing platforms in a master-worker fashion. Divisible load applications are similar to many so-called "embarrassingly parallel applications", but we use the "divisible load" term 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.

Many real-world applications fit the *divisible load* model, including pattern searching applications in computational biology [25], video compression applications [33, 2], volume rendering applications that are used for scientific computing and biomedicine [34, 45, 15, 9, 20, 42, 15], and even Data mining applications [4, 23, 17]. These applications have different characteristics, in terms of total running time, total data size, and computation-communication ratio (which we denote by r throughout). To get an idea of this diversity, we conducted simple experiments for three specific applications: (1) HMMER [25], a bioinformatics application that compares a given protein profile to a database using a Hidden Markov Model technique (we used the example profile globin.hmm generated from the HMMER tutorial dataset against the nr database from the National Center for Biotechnology Information [35]); (2) MPEG4 compression, using the divx4 library and mencoder frontend [33] to compress an MPEG2 source file (A more common example would be compressing an AVI source, but we used MPEG2 because it is simpler to divide the load, although the general behavior is similar.); and (3) VFleet [41], a volume rendering application (we rendered the 256x256x167 "bigbrain" dataset from the VolPack site [44], scaled to 512x512x334, which is more representative of today's volume rendering requirements).

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 this data for the Data Mining application presented in [40]. 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

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 ($\frac{max-min}{mean}$).

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. While at first glance 10% may seem relatively low, it can cause significant load imbalance. The last column of Table 1 shows the value $(max - min)/mean$, where max and min are the maximum and minimum running time for each unit of load, respectively, and $mean$ is the mean running time. We see that for MPEG4 compression, the slowest work unit may take 30% longer than the fastest unit, and for HMMER, the ratio is 2700%. We can then conclude that, for some divisible load applications, using the mean load unit execution time as a prediction of the compute time of a unit of load may lead to considerable performance prediction errors. In fact, it is likely that no good prediction can be performed and that a scheduling algorithm would have to be designed to tolerate such uncertainty.

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. While the divisible nature of the load makes DLS more tractable than other scheduling problems (e.g., ones with fixed-size tasks) and some optimality results are known in a few cases, many challenging issues must be addressed for efficient scheduling [8]. The first proposed DLS algorithms were *One-Round* algorithms, so called because they assign each worker exactly one chunk of the load. These algorithms were studied for many platform topologies (e.g., Linear Networks, Single-level Trees, Meshes, Hypercubes) and we refer the reader to the survey in [8] 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 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 algorithm in [10] proposes a multi-round algorithm that assumes purely linear communication and computation costs. [47] 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 [48] computes a near-optimal number of rounds with affine costs and is applicable to heterogeneous platforms, which represents a major advance for using multi-round DLS in practice.

Finally, the recently proposed RUMR algorithms [48] extends UMR and attempts to mitigate the effects of *uncertainty* on chunk communication and computation times, which can be caused by the platform (e.g., when resources are non-dedicated and time-shared) or by the application (i.e., when the computation is data-dependent). 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 Factoring approach [26, 24, 27].

In this work we focus on Multi-Round algorithms, as they achieve better performance than One-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 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 the recent surveys [11, 37], to the special issue of the Cluster Computing journal [1], and to the Web page collecting related literature [38] for complete details about DLS research.

3 The APST-DV Software

3.1 APST Background

APST is a Grid application execution environment originally targeted to "Parameter Sweep Applications" that consist of many independent tasks. The APST software was designed with the goal of fully automated and transparent deployment of applications on Grid infrastructures, 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 no modification of the application is required.

APST relies on deployed services to access and monitor storage, compute, and network resources. Compute resources can be accessed via the Globus Toolkits [22], or via Ssh as a default. APST can use the above mechanisms to access batch-scheduled resources via PBS [36], LoadLeveler [31], Sun Grid Engine [39], Condor [30], etc. APST can read, copy, transfer, and store application data among storage resources with GASS [18], GridFTP [3], or SRB [7]. As defaults, APST can also use Scp or FTP. Finally, APST can obtain static and dynamic information from the MDS [16], NWS [46], and Ganglia [19] information services. APST also learns about the performance of available resources by keeping track of their past performance when computing application tasks or transferring application data.

APST is currently used in production for a number of applications, including the MCell neuroscience application [12], the Encyclopedia of Life (EOL) bioinformatics application [29], the Vizport visualization portal [43], and the discrete-event simulation application SIMGRID [28]. We refer the reader to [14, 5] 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 small identical pieces) are bound to achieve poor performance. So while APST implements good scheduling algorithms and 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 success 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 several 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 some of the 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 [5] for a complete 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). The `uniform` method divides the input every `stepsize` units, where the unit is specified by the `steptype` attribute, from a starting offset of `start`. In the sample XML this is the method used and 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). The `callback` method allows the user to provide her own script/program that computes the closest legal boundary for a division near a specified offset. Finally, the `index` method allows the user to provide an "index file" that lists all valid division boundaries. Note that APST-DV divides the load on-the-fly, thereby avoiding creating a prohibitive number of files for each individual chunk.

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

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

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 3.4.

3.4 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 [16], NWS [46], Glanglia [19]. Some of this information can be dynamic and must be retrieved periodically. The advantage of this approach is that it is light-weight. The main drawback is that it is difficult to obtain accurate estimates of computation and transfer times. The second approach is to just ob-

serve 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 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 [21]. 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. Our probes are not part of the actual load but rather a 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). The input file used for probing is specified by the `probefile` attribute in the XML specification of a divisible load application (see Figure 1). The work in [21] proposes sophisticated probing strategies that overlap probing with application execution, which we will explore in future work to further increase performance.

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

3.5 Scheduling in APST-DV

The current APST-DV prototype implements the following four 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)** [49] – 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** [27] – 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 and also observes chunk execution times throughout application execution to refine its estimates of worker speeds. The factoring method was specifically designed to deal with uncertainty in computation times: application execution ends with small chunks, which make it easier to do load-balancing. However, Factoring was not designed to maximize overlap of communication and computation.
>
> **Robust Uniform Multi-Round (RUMR)** [48] – 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 [48] 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 [49] it was shown that UMR outperforms competing multi-round algorithms and largely outperforms SIMPLE-n. In [48] 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). Our synthetic application reads in an input file, does some floating point operations on the bytes read in a loop, l times. 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 workload). In our experiment we experiment with several values for l and on different platforms, which leads to different values for r, and with $\gamma = 0\%$ and $\gamma = 10\%$ to look at applications that do not or do exhibit inherent uncertainty (these values are the two extremes of the range of values with have seen with real-world applications).

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 [39] and PBS [36] batch schedulers. The APST daemon and initial input for the divisible load application were located in the Grid and Research and Innovation Laboratory (GRAIL) at UCSD, about 1/2 mile from SDSC.

Our focus on platforms whose processors are dedicated during application execution is representative of most production environments. Explicitly accounting for delays incurred to acquire these resources (e.g., batch queue waiting time) is difficult. In our experiments we just wait for all batch resources to be allocated. This is so that we can ignore the effects of queue waiting time and perform deterministic and fair comparisons among experiments. In practice, the APST-DV implementation just treats a set of batch-scheduled clusters as a dynamically growing and shrinking pool of processors.

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 3.5, 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. By contrast, our workers are ded-

icated, as they are batch-scheduled, and do not lead to (significant) uncertainty. This is required to enable reproducible, scientifically valid, *real-world* experiments (in our previous work we experimented with uncertainty in *simulation* [48]). Consequently, the only significant source of uncertainty in our setup is the application itself which allows us to control our experiments. Another source of uncertainty was the network, but we witnessed stable network conditions during our experiments.

Note that some of our DLS algorithms would also mitigate the effect of uncertainty due to resource availability fluctuations. But due to the specific statistical property of these fluctuations, results may differ from the ones presented in this paper. It is outside the scope of this paper to study the impact of the specific properties of uncertainty on divisible load scheduling.

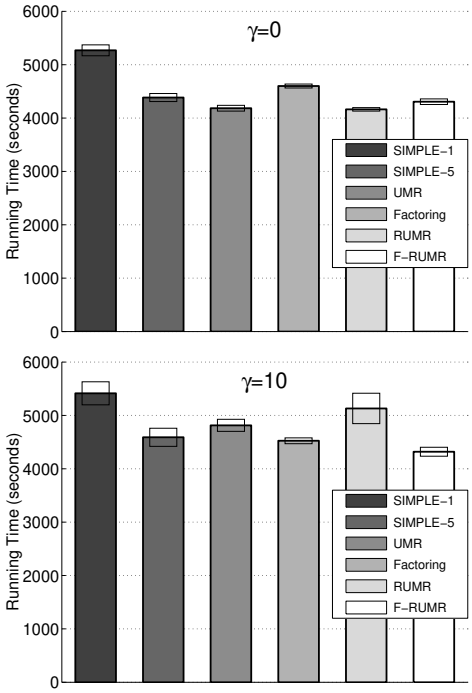

Figure 2: DAS-2, 16 nodes

4.2 Experimental Results

We ran APST-DV with all the DLS algorithms described in Section 3.5, back-to-back. Each data point corresponds to an average over 4 distinct runs. (Note that we observed small variations in our experiments, on the order of a few percents as seen in error bars in our results). Each application run lasted between 70 minutes and 110 minutes, depending on the resources and the scheduling algorithm used.

DAS-2, 16 nodes, r = 35, γ = 0, 10 – We first ran our application only on the DAS-2 cluster. 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 almost no uncertainty and RUMR degenerates to pure UMR as there is no second phases) 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 (6% slower), while SIMPLE-1 is 27% slower. The factoring algorithms are roughly 10% slower than UMR/RUMR. These results confirms some of the simulation results presented in [49, 48].

For $\gamma = 10$ something surprising happens. With more uncertainty we would expect Weighted Factoring to perform relatively better than UMR, which is the case (e.g., Weighted Factoring is about 7% faster than UMR). However, the simulation results in [48] indicate that RUMR should outperform Weighted Factoring as it is striving to both overlap communication and computation, and to mitigate the effects of uncertainty. However, 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 [48] assumes that the value for γ is known in advance and, using this value, predetermines 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 the 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 phases, meaning that factoring is

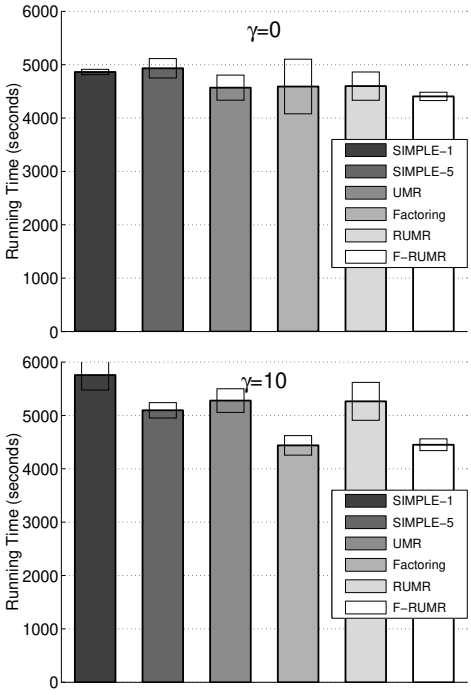

Figure 3: Meteor, 16 nodes

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 uncertainty could be learned from past application executions). However, 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 = 47, 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 10% and 12% 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 118 kB/sec compared to 94 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. The SIMPLE-n algorithms do not perform well, as expected.

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 (both 19% slower) suffer from the same problems as discussed above for the DAS-2 experiments. But, importantly, Fixed RUMR leads to 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 SIMPLE-1 and SIMPLE-5 have poor performance (28% and 18% 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 (27% and 11% slower).

4.3 Discussion

From the experimental results above (we also ran experiments with different subsets of our clusters and different values of l 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 24% and 11% 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

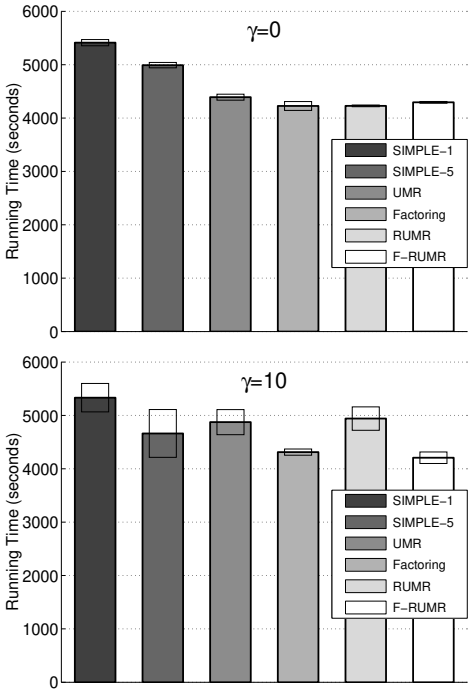

Figure 4: Meteor+DAS-2, 16 nodes

is poor when uncertainty becomes significant (15% 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 [48] 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 should be used by APST-DV users.

5 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*. These applications are commonplace, as seen in Section 2.1, and to the best of our knowledge our work provides the first generic software environment to deploy them on current distributed computing platforms. APST-DV embeds a scheduler that currently implements four Divisible Load Scheduling (DLS) algorithms. We have demonstrated the use of APST-DV and 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 [48] is the most effective across the board provided that there is a better mechanism for switching between the two phases of its execution. For now we have given a simple version, Fixed-RUMR, that performs very well in practice, and should be used by APST-DV users.

We will extend this work in several directions. First, 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 (note that our version of Weighted Factoring performed such adaptation and thus has somewhat of an unfair advantage over RUMR). The results in this paper validate our prototype implementation of APST-DV, and we will release the software as part of the APST 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 Meta-computing. *Linux Journal*, 116:8, 2003.

[3] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, and L. Liming. GridFTP: Protocol Extension to FTP for The Grid. Grid Forum Internet-Draft, March 2001.

[4] N. Amano, J. 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.

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

[6] 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.

[7] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In *Proceedings of CASCON'98, Toronto, Canada*, November 1998.

[8] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: results and open problems. Technical Report RR-2003-41, LIP, École Normale Supérieure de Lyon, September 2003.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] Y.-J. Chiang, R. Farias, C. T. Silva, and B. Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proceedings of the IEEE Symposium on Parallel and Large-data Visualization and Graphics*, pages 59–66, 2001.

[16] 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.

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

[18] I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the Sixth workshop on I/O in Parallel and Distributed Systems*, May 1999.

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

[20] 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.

[21] 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.

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

[23] 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.

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

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

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

[27] 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.

[28] 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.

[29] 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.

[30] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceesings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[31] IBM LoadLeveler User's Guide, 1993. IBM Corporation.

[32] MCell Webpage. `http://www.mcell.cnl.salk.edu/`.

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

[34] 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.

[35] National Center for Biotechnology Information (NCBI). `www.ncbi.nlm.nih.gov/`.

[36] The Portable Batch System Webpage. `http://www.openpbs.com`.

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

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

[39] Sun Grid Engine. `http://gridengine.sunsource.net/`.

[40] 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.

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

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

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

[44] Volpack volume rendering package. `http://graphics.stanford.edu/software/volpack/`.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.