Dharma: a Grid-enabled Domain-specific Metaware for Hydrology

Daniel Andresen*, Mitchell Neilsen*, Gurdip Singh*, Prasanta Kalita†
*Department of Computing and Information Sciences
234 Nichols Hall, Kansas State University
{dan, neilsen, singh}@cis.ksu.edu
Office: (785)532-6350, Fax: (785)532-7353
†Department of Agricultural Engineering
University of Illinois, Urbana-Champaign

Abstract

The DHARMA domain-specific middleware system is intended to allow hydrologic field engineers to tackle water-management problems on a scale previously impossible without sophisticated computational management systems. DHARMA provides automatic data acquisition via the Internet; data fusion from online, local, and cached resources; smart caching of intermediate results; parallel process execution; automatic transformation and piping of data between different hydrologic simulation models; and interfaces with existing metacomputing systems. Our target watershed model, WEPP, is limited to very small watersheds with current computer technology. A revolutionary change in hydrologic modeling on the watershed scale is being brought about by applying various watershed models to large watersheds, such as the Lake Decatur Watershed which covers 925 square miles.

Even with unlimited hardware, the current version of WEPP can not be applied directly to the Lake Decatur Watershed because WEPP is limited to a maximum of 75 hillslopes. To address this issue, we have integrated support for heterogeneous simulation models in our software; for example, we can use WEPP to model individual hillslopes and SITES to model the connecting channels (reaches) and dams structures, playing to the strengths of both models.

In this paper, we discuss the evolving software architecture of DHARMA and its interaction with the cluster and Grid infrastructures. We also present experimental results showing the performance of DHARMA using Globus over three Beowulf clusters.

Keywords: *distributed computing, hydrology, watershed, XML*

1 Introduction

The DHARMA project attacks two fundamental problems in today's hydrological research environment, particularly in the biological and environmental sciences. First, the inability of researchers to perform even simple investigations due to the inaccessibility and essential difficulty in acquiring and utilizing the necessary data. Often the data and simulation models are available via the Internet, but through a combination of obscurity, incompatibility, and inefficiency are essentially unusable. Second, local computing resources are inadequate to support modeling systems at the level desired, but interfacing to remote resources can require complex software installations and procedures. DHARMA addresses these problems through significantly easier access to the computational power and data acquisition capabilities of the Internet.

The objective of the DHARMA project is to expand the applicability of the WEPP (Water Erosion Prediction Project) hillslope model and the SITES (Water Resource Site Analysis) dam simulation to large watersheds, specifically applying the extended model to the 925 sq. mile Lake Decatur watershed in Illinois.

DHARMA has the following major points of functionality: automatic data acquisition of geotemporal climatic data from online databases; data merging necessary for the computation to occur, from online, local, and cached resources; automatic transformation and piping of data between heterogeneous hydrological simulations; smart caching of intermediate results to allow for reuse in future simulation cycles; task graph acquisition from the UI layer, and optimization through application-specific knowledge and dynamic results caching strategies; and interfacing with computing resource managers, such as Globus and Condor [5, 7].

To address these issues, we:

- use the eXtensible Markup Language (XML) and DTDs as a lingua franca between simulations, remote data sources, and DHARMA components.
- realize our primary goal is flexibility and ease in data acquisition and data transfer between simulations, which cannot be sacrificed for utility-limiting performance enhancements.
- allow DHARMA to have multiple interfaces to existing metacomputing systems, and use DHARMA to package up the computation into a tidy set for computation by these systems.
- develop heuristics for scheduling task graphs, interfaces for the various resources, and application-specific caching techniques.

The paper is organized as follows: Section 2 discusses the DHARMA software architecture and various implementation issues relating to the Grid. Section 3 presents our experimental results, and Section 4 discusses related work and conclusions, as well as future directions.

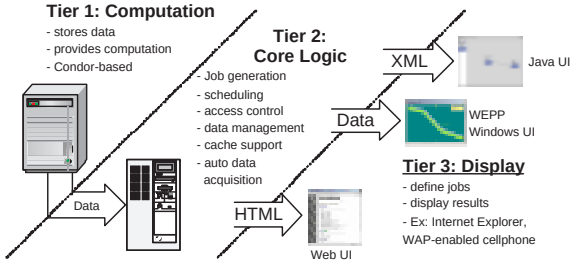

Figure 1. DHARMA layers diagram.

2 DHARMA Architecture

We now give an overview of the DHARMA architecture, as illustrated in Figure 1. Our architecture generally follows the standard n-tier model with a user interface, task logic, and, differing from the standard three-tier model, a cluster-based computation/data layer. This includes the multiple user interfaces, the job manager, automated data acquisition, and execution engine.

The Dharma UI There are three possible interfaces that can be used to run a simulation in our system. The three interfaces are the WEPP Windows Interface, the Dharma Web Interface and the Drag and Drop Interface. Each of these interfaces retrieve data from the user and interact with the middleware in a different manner. The primary interface that will be used when the project is fully functional will be the Drag and Drop Interface. The other two interfaces will still be supported at the end of the project. These interfaces have been explained in more detail in previous publications [2].

The Drag and Drop Interface (DND) is an interactive interface that allows the user to create a diagram that visually represents a simulation run (Figure 2). The interface is deployed on client machines using Java Web Start and written in Java. The Drag and Drop interface is a highly extensible user interface that allows the Dharma project to function for many different simulations. The only modifications that are necessary to extend the interface are an XML document detailing the inputs necessary for the creation of a dialog box for that simulation.

Figure 2. Dharma Drag-and-Drop Interface with SITES and WEPP simulations.

2.1 DHARMA middleware

The middleware component of the DHARMA system provides task scheduling, dependency management, simulation-specific data preparation and interpretation, and fail-safe task execution.

A DHARMA job is modeled as a directed, acyclic graph. Nodes in the graph correspond to individual simulation instances. One describes the structural layout of a job with XML documents. For instance, we detail a very small watershed's specification in Figure 3 that illustrates the use of multiple simulation models and inter-task dependencies.

Task Management Once a job has been submitted to the DHARMA system, it is wrapped within a job executer. The job executer then partitions the job into multiple smaller jobs based on inter-task dependencies. The sub-job is represented as an independent job allowing us to run DHARMA within DHARMA giving us parallelism across multiple clusters during a job execution. Each sub-job is associated with a new task and

```
<job>
  <id>0</id>
  <task>
    <id>0</id>
    <type>wepp.simulation.hillslope</type>
    <resource>
      <id>0</id>
      <type>wepp.types.climate</type>
      <url>file://grid/dharma/0/shed1.cli</url>
      <result>false</result>
    </resource>
    ...
  </task>
  ...
  <task>
    <id>3</id>
    <type>sites.simulation.junction</type>
    <dependency dependency="0"/>
    <dependency dependency="1"/>
    <dependency dependency="2"/>
  </task>
</job>
```

Figure 3. XML specification for a job

added into pools of runnable and non runnable tasks based on the their inter-dependencies.

For each available cluster, a sub middleware server is started along with a specified number of clients. All clients are able to communicate with the server through RMI. Upon retrieval of a runnable task, or sub-job, from the parent middleware server, a job exectuer constructs wrappers around each individual task and adds them to a pools of runnable and non-runnable tasks. Client processes (each a separate JVM on one of a cluster's machines) then make a API calls to the scheduler component, so that they may retrieve individual tasks for execution.

Upon retrieval of a runnable task t_0 from the scheduler host, a client examines the type tag of t_0 and instantiates a "task executor." This task executor is responsible for reformatting DHARMA's idealized XML representation of hydrological data into the native representation of t_0's model.

After data have been prepared for the model, the task executor submits a request for external program invocation to a DHARMA "execution manager." The abstraction of the execution manager allows the task executor to assume that a simulation has been run, without caring whether this happens as a local process, a Globus job, or some other means of spawning an external process. We discuss this concept more fully in Section 2.4.

When the execution manager reports the model has finished, the task executor interprets the results of the simulation. All relevant data are reformatted as XML and cached in the metadata manager.

Last, the task executor reports back to the scheduling server that t_0 has finished. The scheduler then determines which of t_0's dependent children are now runnable, and moves such tasks to the pool of runnable tasks.

A correctly compiled task is guaranteed to execute and finish regardless of lost clients or any other problems that might arise outside DHARMA's control. Error recovery is handled by the job executers. Each task and sub-job is given a timeout and a maximum of 2 failed executions before the job execution is abandoned.

2.2 DHARMA on the Grid

Although performing medium to large simulations may best be achieved on a local cluster, one wishes to exploit the full power of the supercomputing grid to compute extremely large results.

To adapt DHARMA to such a task, we have adapted DHARMA's notion of resource management to rely upon services offered by the Globus gatekeeper system. We present these modifications in Table 1.

To achieve acceptable computation/communication ratios, DHARMA prefers to schedule interdependent tasks on the same cluster. This is achieved by first applying a partioning algorithm to the original job graph. As discussed earlier, tasks are packaged together into sub-jobs and sent to remote clusters as a unit for computation.

We have chosen to represent an entire partition as a single DHARMA job to avoid the redundancy of creating a second job-managing service to manage these partitions. This allows us to nest an entire instance of the DHARMA system inside the infrastructure of the "task executors".

Selection of the size of partitions is currently done manually. There are two main factors that go in to selecting the size of a partition. One problem that we want to avoid is the situation where a fast cluster is idle because there are no available partitions while a slower cluster is busy processing. Reducing the partition size allows for better load balancing which helps to alleviate the problem. The overhead of starting up a partition is constant relative to the size of the partition, though. Decreasing the partition size will increase the number of partitions that in turn will increase the overall overhead of partitioning. We used experimentation to determine a good partition size to use.

The partition startup cost is due to marshalling of partitions, unmarshalling of partitions, merging of results, and network transfer time. We have seen a slight increase in performance from the new partitioning algorithm, but in the future, we plan on reducing the costs further by pipelining execution of partitions to reduce this overhead. Pipelining will allow a cluster to fetch and setup one partition while executing another parti-

Feature	Local implementation	Grid implementation
Retrieval of results	NFS file system	HTTP
External program execution	Process spawning	Globus job
Metadata registration	Intra-process API calls	RMI/SOAP service

Table 1. Comparison of resources in local/grid modes

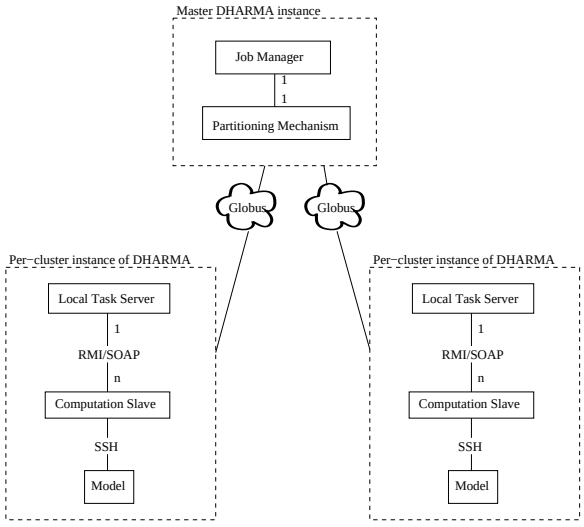

Figure 4. DHARMA in multicluster operation

tion. With this optimization, the overhead of partition startup could be greatly reduced allowing us to create smaller partition sizes.

The execution manager service simply chooses the Globus Gatekeeper as the means of spawning the external "simulation" that is an entire DHARMA system.

2.3 Metadata Manager

In order to make the system more efficient, we cache files that are submitted to us by the users when they submit a job. We also cache the output files that are created by simulation runs. To solve the problem of caching we created the Metadata Manager (MDM). The MDM is responsible for maintaining information on all of the files that have been cached on the Dharma servers.

The MDM stores information on all of the files in the system in a MySQL database. The primary things that are stored by the MDM are file name, file size, checksum of the file, file type, the lat-long box that the file stores data on, location on servers, and other specialized attributes. The manager was designed to be easily extensible. For each type of file in the system there is a Java class that represents the data stored on that file. These classes contain information that is stored about the files. There are also methods in these classes that tell a database how to create tables for this type of file and create queries to get information about these file types.

To access data in the MDM you must create a SOAP request that the MDM will process and return the results of the query. When looking for a file matching some specific parameters, the user will be returned a list of URLs to where the files are located on the Dharma servers. Users of the system are not allowed to directly access the database that stores the information on the files for security reasons. The abstraction added by the MDM provides a security buffer between the metadata and the outside world. This will also allow us to later implement the idea of file ownership.

The Meta Data Manager was a necessary buffer between the metadata and outside world. The main reason for this was security. However, we also had to have an efficient way to store information on files that we have cached so that our system would run efficiently. In our test system, execution times typically drop over 70% compared to uncached execution (1178ms vs. 400ms from job receipt to WEPP startup), indicating the overhead introduced by our system is small and the caching is effective.

2.4 The backend

The DHARMA system employs an abstraction mechanism for process creation, which we term the backend. This affords the middleware task executor a considerable degree of flexibility when spawning models. We have observed the need for submitting process execution requests in the following ways:

1. Local processes on the host running the task executor
2. Processes scheduled within a local cluster, demanding extremely low turnaround time
3. Jobs submitted to the Globus infrastructure for batch execution on remote computing clusters

To accommodate these disparate needs, we developed a generic API—the "execution manager"—for the backend which is ignorant of the actual means used to

schedule a process. The middleware makes all requests for program execution strictly to this abstract API. Providing the varying means of execution then becomes as simple as writing Java classes to implement the execution manager API. To date, we have provided DHARMA with execution managers that leverage local process creation; a high-throughput scheduler which leverages the Secure Shell Server [8]; and a wrapper around the Globus job submission API for moving tasks to remote clusters.

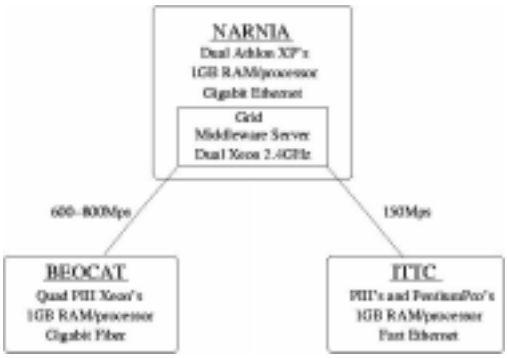

Figure 5. Cluster Configuration

3 Experimental Results

Our experimental results are positive. Recently, we have started running simulations of the Lake Dectaur watershed using data from UIUC containing approximately 3200 hillslopes over its 920 sq. miles. We have also used sample data and randomly generated task graphs with WEPP hillslope leaf nodes and SITES interior nodes for larger test runs. Preliminary runs of the Lake Decatur simulation have been successful - the first time ever that a watershed this large has been modeled using WEPP. We estimate a total of approximately 10,000 hillslopes will be required for a fully-detailed model of the Lake Decatur watershed. The project to gather this data is underway at UIUC, and is anticipated to finish sometime this year.

Overhead for the system is somewhat steep, given its distributed nature and the degree of data conversion/translation necessary for the heterogeneous simulations. There are three clusters including in our simulation runs (Figure 5). We use three Beowulf clusters, each running Globus as an interface to Condor on Linux [5, 9]. “narnia” is primarily composed of 16 dual Athlon MP processors with 1.5-1.8Ghz. processors in addition to 3 dual Pentium III’s all with 1GB RAM per processor, while “beocat” consists of three quad-processor and one dual-processor Pentium III Xeons (approx. 600Mhz.,1GB RAM/processor). The Beowulf clusters are internally connected by gigabit Ethernet, and linked by the campus gigabit Ethernet backbone, which typically has 600-800Mbps. available. The third cluster is at the Information and Telecommunication Technology Center (ITTC) at the University of Kansas, which consists of 8 PIII’s with 1GB each connected by Fast Ethernet internally and 155Mpbs Internet2 to the KSU campus. Currently, our middleware server runs on a dual 2.4GHz Xeon machine with 2GB RAM and our metadata server runs on a dual 1.8GHz Xeon with 4GB RAM.

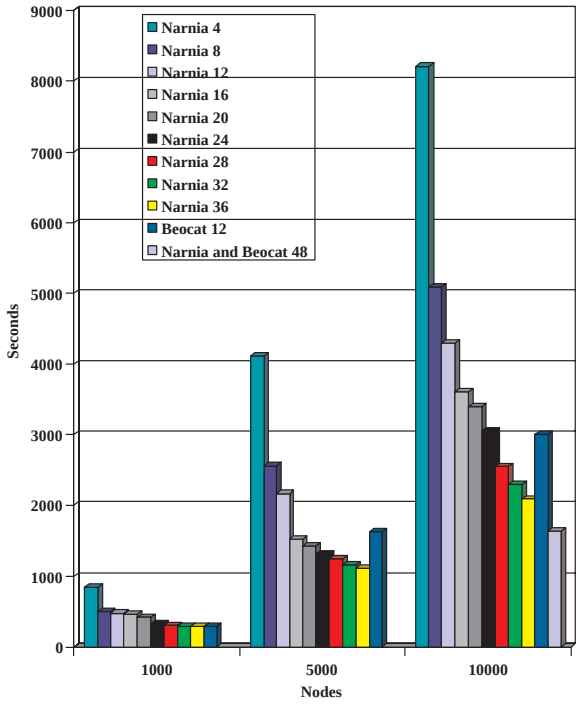

Figure 6. Cluster runtimes.

Tasks consist of a mixture of hillslopes simulated through WEPP, and channels/impoundments simulated via SITES. Each SITES task for reach routing a simple channel takes < 1 s. to run, while each hillslope requires approximately 4 s. of computation. Starting up the client and remote servers under Globus and Condor takes approximately 30 s., and starting up a 10K node job takes approximately 60-90 s. As can be seen in Figure 6, scalability is approximately as expected. Adding the cluster from ITTC in general degraded performance due to the low computation/communication ratio for our tasks. Compute nodes were typically running at 100% CPU utilization, while the master nodes, file server, and database were typically sitting at ¡1% utilization. A typical 10K node run with both clusters takes 2300 s., produces 7,998 files containing 244MB of data, transfers 130MB of information between clusters, and makes over 20,000 RMI calls to the MDM and 86,000 database queries. Numbers for runs up to

100K nodes are proportional to those in Figure 6, with runtimes of 4-5 hours.

A great deal of effort was required to modify our architecture to achieve acceptable performance under a Grid-based system, even given the relatively high bandwidth and low latencies in our test configuration. Initial results treating Globus identically to our high-throughput scheduler (see [2] for details) were atrocious, due to the high overhead and poor computation/communication ratio. The current results still suffer from the effects of the overhead, but are more acceptable due to the node batch partitioning scheme and aggressive data caching. We note that in the future, our scheduler will need to become more sophisticated to make optimal use of local clusters with our high-throughput/low-overhead scheduler for small jobs, while using remote resources for large simulation runs.

4 Related work and conclusions

Presenting a simple interface to the researcher and educator, with its seamless and transparent access to distributed databases, is a difficult technological challenge. Various other distributed computation systems have previously been developed but almost invariably are aimed at automating the distribution of existing scientific programming and data. Systems such as SWEB++, AppLeS, and Globus all will schedule computation quite competently [1, 3, 5]. However, they fail to address the most fundamental difficulty in conducting any sort of operation on distributed data sets – the acquisition and matching of data to the models being used. Furthermore, the interface presented to the user is often more suited to the professional programmer rather than a researcher in the physical sciences. Systems devoted to universal data access, such as the WebHLA and DATORR projects, lack the domain-specific knowledge to acquire and optimally use the necessary hydrological data [6, 4].

In this paper we have discussed the DHARMA system, which, through the use of standards from the computational Grid and the WWW, meets our goals of increasing user capabilities while maintaining simplicity of use. We show how we have implemented Grid integration, multi-level caching, and an extensible set of user interfaces to bring hydrologists into a new era.

In the near future we are extending DHARMA in two directions. First, we are working to incorporate the HEC-RAS river modeling tool. This will significantly increase the accuracy of our channel simulation, and, due to the fact that HEC-RAS allows for backflows where two channels meet, forcing us to extend our dependency graphs to include cycles. Second, we are developing the tools to automatically set the job aggregation size to increase our overall execution efficiency. We also plan to explore using the new Globus toolkit 3.0.

Acknowledgments

This work was supported in part by startup funds from Kansas State University, and NSF ITR-0082667. We also wish to thank Michael C. Hirschi, Ryan Porter, Jesse Greenwald, Dan Lang, Matt Hoosier, Mahesh Kondwilka, T.J. Rothwell, Doug Armknecht, David Sexton, and Bradley Hajek for their efforts and enthusiasm.

References

[1] D. Andresen and T. Yang. Multiprocessor scheduling with client resources to improve the response time of WWW applications. In *Proceedings of the 11th ACM/SIGARCH Conference on Supercomputing (ICS'97)*, pages 92–99, Vienna, Austria, July 1997.

[2] Daniel Andresen, Mitchell Neilsen, Gurdip Singh, and Prasanta Kalita. Domain-specific metaware for hydrologic applications. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 416–421, Cambridge, MA, November 2002.

[3] Fran Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996.

[4] D. Bernholdt, G. Fox, W. Furmanski, B. Natarajan, H. Ozdemir, Z. Ozdemir, and T. Pulikal. WebHLA - an interactive programming and training environment for high performance modeling and simulation. In *Proceedings of the DoD HPC 98 Users Group Conference*, Houston, TX, June 1998.

[5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[6] G. C. Fox, W. Furmanski, G. Krishnamurthy, and H. T. Ozdemir. Using WebHLA to integrate HPC FMS modules with Web/commodity based distributed object technologies of CORBA, Java, COM and XML.

[7] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computer Systems*, pages 104–111, San Jose, California, June 1988. IEEE.

[8] OpenSSH, July 2003. http://www.openssh.com/.

[9] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, August 1995.