Application-Aware Scheduling of a Magnetohydrodynamics Application in the Legion Metasystem

Holly Dail* Graziano Obertelli*

Francine Berman* Rich Wolski† Andrew Grimshaw‡

* Computer Science and Engineering Department
University of California, San Diego
[hdail, graziano, berman] @cs.ucsd.edu

† Department of Computer Science
University of Tennessee
rich@cs.utk.edu

‡ Department of Computer Science
University of Virginia
grimshaw@virginia.edu

Abstract

Computational Grids have become an important and popular computing platform for both scientific and commercial distributed computing communities. However, users of such systems typically find achievement of application execution performance remains challenging. Although Grid infrastructures such as Legion and Globus provide basic resource selection functionality, work allocation functionality, and scheduling mechanisms, applications must interpret system performance information in terms of their own requirements in order to develop performance-efficient schedules.

We describe a new high-performance scheduler that incorporates dynamic system information, application requirements, and a detailed performance model in order to create performance efficient schedules. While the scheduler is designed to provide improved performance for a magneto hydrodynamics simulation in the Legion Computational Grid infrastructure, the design is generalizable to other systems and other data-parallel, iterative codes. We describe the adaptive performance model, resource selection strategies, and scheduling policies employed by the scheduler. We demonstrate the improvement in application performance achieved by the scheduler in dedicated and shared Legion environments.

This research was supported in part by DARPA Contract#N66001-97-C-8531, DoD Modernization Contract 9720733-00, and NSF/NPACI Grant ASC-9619020

1. Introduction

Computational Grids [7] are rapidly becoming an important and popular computing platform for both scientific and commercial distributed computing communities. Grids integrate independently administered machines, storage systems, databases, networks, and scientific instruments with the goal of providing greater delivered application performance than can be obtained from any single site. There are many critical research challenges in the development of Computational Grids as an effective computing platform. For users, both performance and programmability of the underlying infrastructure are essential to the successful implementation of applications in Grid environments.

The **Legion** Computational Grid infrastructure [11] provides a sophisticated object-oriented programming environment that promotes application programmability by enabling transparent access to Grid resources. Legion provides basic resource selection, work allocation, and scheduling mechanisms. In order to achieve desired performance levels, applications (or their users) must interpret system performance information in terms of requirements specific to the target application. **App**lication **Le**vel **S**cheduling (**AppLeS**) [3] is an established methodology for developing adaptive, distributed programs that execute in dynamically changing and heterogeneous execution settings. The ultimate goal of this work is to draw upon the AppLeS and Legion Computational Grid research efforts to design an adaptive application scheduler for regular iterative stencil codes in Legion environments.

We consider a general class of regular, data-parallel stencil codes which require repeated applications of relatively constant-time operations. Many of these codes have the following structure:

```
Initialization

Loop over an n-dimensional mesh

Finalization
```

in which the basic activity of the loop is a stencil based computation. In other words the data items in the n-dimensional mesh are updated based on the values of their nearest neighbors in the mesh. Such codes are common in scientific computing and include parallel implementations of matrix operations as well as routines found in packages such as ScaLAPACK [18].

In this paper we focus on the development of an adaptive strategy for scheduling a regular, data-parallel stencil code called PMHD3D on the Legion Grid infrastructure. The primary contributions of this paper are:

- We describe an adaptive performance model for PMHD3D and demonstrate its ability to predict application performance in initial experiments. The performance model represents the application's requirements for computation, communication, overhead, and memory, and could easily be extended to serve more generally as a framework for regular iterative stencil codes in Grid environments.

- We couple the PMHD3D performance model with resource selection strategies, schedule selection policies, and deployment software to form an AppLeS scheduler for PMHD3D.

- In order to satisfy the requirements of the PMHD3D performance model we implement and utilize a new *memory sensor* as part of the Network Weather Service (NWS)[22]. The sensor collects measurements and produces forecasts of the amount of free memory available on a processor.

- We demonstrate the ability of the AppLeS methodology to provide enhanced performance for the PMHD3D application, using the Legion software infrastructure as a platform for high-performance application execution.

In the next section we discuss the structure of the target application and the environment that we used as a test-bed. In Section 3, we discuss the AppLeS we have designed for PMHD3D and provide a generalizable performance model. Section 4 provides experimental results and demonstrates performance improvements we achieved via AppLeS using Legion. In Sections 5 and 6 we review related work and investigate possible new directions, respectively.

2. Research Components: AppLeS, NWS, PMHD3D and Legion

In order to build a high-performance scheduler for PMHD3D we leveraged application characteristics, dynamic resource information from NWS, the AppLeS methodology, and the Legion system infrastructure. In this section we explain each of these components in detail.

2.1. AppLeS

The AppLeS project focuses on the development of a methodology and software for achieving application performance via adaptive scheduling [1]. For individual applications, an *AppLeS* is an agent that integrates with the application and uses dynamic and application-specific information to develop and deploy a customized adaptive application schedule. For structurally similar classes of applications, an *AppLeS template* provides a "pluggable" framework which comprises a class-specific performance model, scheduling model, and deployment module. An application from the class can be instantiated within the template to form a performance-oriented self-scheduling application targeted to the underlying Grid resources.

AppLeS schedulers often rely on available tools in order to deploy the schedule or to gather information on resources or environment. AppLeS commonly depends on the Network Weather Service (NWS) (see Section 2.4) to provide dynamic predictions of resource load and availability. Together, AppLeS and the Network Weather Service can be used to adapt application performance to the deliverable capacities of Grid resources at execution time. In this project AppLeS uses Legion to execute a schedule and the Internet Backplane Protocol (IBP) [13] to effectively cache the data coming from NWS.

2.2. PMHD3D

The target application for this work, PMHD3D [12, 15], is a magnetohydrodynamics simulation developed at the University of Virginia Department of Astronomy by John F. Hawley and ported to Legion by Greg Lindhal. The code is an MPI FORTRAN stencil-based application and shares many characteristics with other stencil codes. The code is structured as a three-dimensional mesh of data, upon which the same computation is iteratively performed on each point using data from its neighbors. PMHD3D alternates between CPU-intensive computation and communication (between "slab" neighbors and for barrier synchronizations).

At startup PMHD3D reads a configuration file that specifies the problem size and the target number of processors. Since the other two dimensions are fixed in PMHD3D's three-dimensional mesh, we refer to the height of the mesh

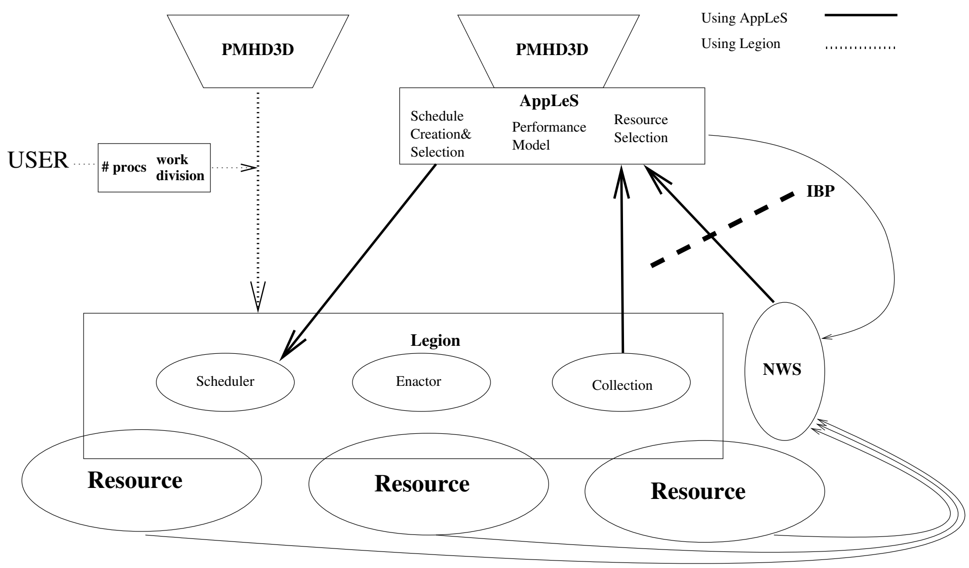

Figure 1. PMHD3D run-time scenarios with and without AppLeS.

as the *problem size*. In order to allocate work among processors in the computation the mesh is divided into horizontal *slabs* such that each processor receives a slab. For load balancing purposes each processor can be assigned a different amount of work (by dividing the work into slabs of varying height). The AppLeS scheduler determines the optimal height of each slab depending on the raw speed of the processor and on NWS forecasts of CPU load, the amount of free memory, and network conditions. AppLeS is dynamic in the sense that the data used by the scheduler is computed and collected just before execution, but once the schedule is created and implemented, the execution currently proceeds without interaction with the AppLeS.

2.3. Legion

Legion, a project at the University of Virginia, is designed to provide users with a transparent, secure, and reliable interface to resources in a wide-area system, both at the programming interface level as well as at the end-user level [9, 14]. Both the programmer and the end-user have coherent and seamless access to all the resources and services managed by Legion. Legion addresses challenging issues in Computational Grid research such as parallelism, fault-tolerance, security, autonomy, heterogeneity, legacy code management, resource management, and access transparency.

Legion provides mechanisms and facilities, leaving to the programmer the implementation of the policies to be enforced for a particular task. Following this idea, scheduling in Legion is flexible and can be tailored to suit applications with different requirements. The main Legion components involved in scheduling are the *collection*, the *enactor*, the *scheduler*, and the *hosts* which will execute the schedule [5]. The collection provides information about the available resources and the scheduler selects the resources to be used in a schedule. The schedule is then given to the enactor, which contacts the host objects involved in the schedule and attempts to execute the application. This scheme provides scheduling flexibility; for example, in case of host failures, the enactor can ask the scheduler for a new schedule and continue despite the failure, the collection can return subsets of the resources depending on the user and/or the application, or the hosts can refuse to serve a specific user.

Legion currently provides default implementations of all the objects described herein. Moreover, new objects can be developed and used rather than the default ones. Note that the PMHD3D AppLeS is developed "on top" of Legion, and uses default Legion objects. We would expect the performance improvement for such a code to conservatively bound from below that which would be achievable if the AppLeS were structured as a Legion object. We plan to eventually develop the AppLeS described here as a Legion scheduling object for a class of regular, iterative, data-parallel applications.

2.4. Network Weather Service

The Network Weather Service [17, 22] is a distributed system that periodically monitors and dynamically forecasts the performance various network and computational resources can deliver. NWS is composed of sensors, memories and forecasters. *Sensors* measure the availability of the resource, for example CPU availability, and then record the measurement in a NWS *memory*. In response to a query, the NWS software will return a time series of measurements from any activated sensor in the system. This time series can then be passed to the NWS *forecaster* which predicts the future availability of the resource. The forecaster tests a variety of predictors and returns the result and expected error of the most accurate predictor. To obtain better performance for PMHD3D we developed a **memory sensor** that measures the available free memory of a machine. The sensor has been extended and is now part of NWS.

2.5. Interactions Among System Components

PMHD3D can directly access Legion's scheduling facilities or can use AppLeS to obtain a more performance-efficient schedule. Figure 1 shows the interactions among components in each of these scenarios. The dotted line represents the scheduling of a PMHD3D run without AppLeS facilities: the user supplies the number of processors, the processor list, and the associated problem size per processor and the rest of the scheduling process is supplied by a default scheduler within the Legion infrastructure.

When the application uses AppLeS for scheduling, the interactions among components can instead be represented by the solid lines in Figure 1. In this case the user supplies only the problem size of interest. AppLeS collects the list of available resources from the environment (via the Legion collection object or, in our case, via the Legion context space), and then queries NWS to obtain updated performance and availability predictions for the available resources. As the figure shows, AppLeS collects the NWS predictions as an IBP client: the predictions are pushed into the IBP server by a separate process.

AppLeS then creates a performance-promoting adaptive schedule and asks the Legion scheduler to execute it. The schedule is adaptive because AppLeS assigns a different amount of work to each processor depending on their predicted performance. As is suggested by the figure, the PMHD3D AppLeS is built on top of Legion facilities. A future goal is to integrate the AppLeS as an alternative scheduler in Legion for the class of regular, data-parallel, stencil applications.

3. The PMHD3D AppLeS

The general AppLeS approach is to create good schedules for an application by incorporating application specific characteristics, system characteristics, and dynamic resource performance data in scheduling decisions. The PMHD3D AppLeS draws upon the general AppLeS methodology [3] and the experience gained building an AppLeS for a structurally similar Jacobi-2D application [2].

Conceptually, the PMHD3D AppLeS can be decomposed into three components:

- a **performance model** that accurately represents application performance within the Computational Grid environment;
- a **resource selection strategy** that identifies potentially performance-efficient candidate resource sets from those that are available at run time;
- a **schedule creation and selection strategy** that creates a good schedule for each of the various candidate resource sets and then selects the most performance-efficient schedule.

The overall strategy and organization of the scheduler will be discussed here but the details of each component are reserved for the following sections.

An accurate performance model (Section 3.1) is fundamental for the development of good schedules. The performance model is used in two important ways, the first of which is to guide the creation of schedules for specific resource sets. For example, load balancing is a necessary condition developing an efficient schedule but is difficult or impossible to achieve without an estimate of the relative costs of computation on various resources. An accurate performance model is also necessary for selection of the highest performance schedule from a set of candidate schedules.

The resource selection strategy (Section 3.2) produces several orderings of available resources based on different concepts of "desirability" of resources to PMHD3D. Our definitions of desirability incorporate Legion resource discovery results, dynamic resource availability from NWS, dynamic performance forecasts from NWS, and application-specific performance data for each resource. Once complete, the ordered lists of resources are passed on to the schedule creation and selection component of the AppLeS.

The schedule creation step (Section 3.3) takes the proposed resource lists and creates a good schedule for each based on the constraints the system and application impose. System constraints are characteristics such as available memory of the resources while the application constraints are characteristics such as the amount of memory required for the application to remain in main memory. Once all schedules have been created the performance model is used to select the highest performance schedule (the one in which the execution time is expected to be the lowest).

The decomposition of the scheduling process into these disjoint steps provides an overly simplistic view of the interactions between steps. In reality the scheduling process

```
Rset = getResourceSet()                          \\ Available resources obtained from Legion
NWS_data = NWS(Rset)                             \\ NWS forecasts of resource performance
C = getScheduleConstraints()                     \\ Obtain scheduling constraints for simplex
for (balance = {0, 0.5, 1})                      \\ Select for CPU power, connectivity, both
  S = sort(Rset, balance, maxP)                  \\ Returns list of hosts sorted by desirability
  for (n = 2..maxP)                              \\ Searching for correct number of processors
     sched = findSched(n, S, NWS_data, C)        \\ Use simplex to find schedule on S using C
     while (sched is not found)                  \\ Simplex was unsolvable with S and C
       "Schedule constraints are too restrictive"
        relaxConstraints (C)                     \\ More schedule flexibility, more possible error
        sched = findSched(n, S, NWS_data, C)     \\ Try to find schedule again
     endwhile                                    \\ Found a feasible schedule
     if (cost(sched) < best)                     \\ If best one so far keep it, else throw away
        best = sched
     endif
  endfor
endfor
run(best)                                        \\ Best schedule found, run it
```

Figure 2. PMHD3D AppLeS pseudo-code.

requires more complicated interactions. To accurately represent the true interaction of the scheduling components we present a pseudo-code version of the PMHD3D AppLeS strategy in Figure 2. The steps shown in Figure 2 will become clearer in the following sections.

3.1. Performance Model

The goal of the performance model is to accurately predict the execution time of PMHD3D. Since the run-time may vary somewhat from processor to processor, we take the maximum run-time of any processor involved in the computation as the overall run-time. During every iteration each processor computes on its slab of data, communicates with its neighbors, and synchronizes with all other processors.

Formally, the running time for processor i is given by:

$$T_i = Comp_i + Comm_i + Over_i$$

where $Comp_i$, $Comm_i$ and $Over_i$ are the predicted computation time, the predicted communication time, and the estimated overhead for P_i, respectively.

Computation time is directly related to the units of work assigned to a processor (in other words the height of the slab) and to the speed of that processor. The computation time for P_i is:

$$Comp_i = \frac{x_i * BM_i}{Avail_i}$$

where x_i is the amount of work allocated to processor P_i (dynamically determined by the scheduling process), BM_i is a benchmark for the application-specific speed of P_i's processor configuration, and $Avail_i$ is a forecast of the CPU load on processor P_i (obtained from dynamic NWS forecasts). To obtain the benchmarks, we run PMHD3D on dedicated machines with various problem sizes and variable number of hosts. Execution times were proportional to problem size and are given in terms of seconds per point on each platform.

Communication time is modeled as the time required for transferring data to neighboring processors across the available network. This represents communication for all iterations and accounts for both the time to establish a connection and the time to transfer the messages. To simplify the communication model, we have not attempted to directly predict synchronization time or the time a processor waits for a communication partner. We hope instead to capture the *effect* of these communication costs in our estimate of overhead costs, which we discuss shortly. Communication time is then:

$$Comm_i = MB/(b_{i,i+1} + b_{i,i-1}) + M * (l_{i,i+1} + l_{i,i-1})$$

where MB is the total megabytes transfered, M is the number of messages transfered, and b_{ij} and l_{ij} are predictions of available bandwidth and latency from P_i to P_j, respectively. Predictions of available bandwidth and latency between pairs of processors are obtained from dynamic NWS forecasts. To provide an estimate of the number of messages transferred (M) and the megabytes transferred (MB) we examined post-execution program performance reports provided by Legion. For a variety of problem sizes and resource set sizes the number of megabytes transferred varied by less than 5% so we used an average value for all runs. Data transfer does not significantly vary with problem size because the problem size affects only the height of the grid while the decomposition is performed horizontally. Data transfer costs also do not vary with number of processors because each processor must communicate with only its neighbors, regardless of the total number of processors. Although the number of messages transferred varied more significantly from run-to-run we also used an average value for this variable. This approximation did not adversely affect our scheduling ability in the environments we tested; in cases where communication costs are more severe a model could be developed to approximate the expected number of messages transferred.

The **overhead** factor $Over_i$ is included in the performance model to capture application and system behavior that cannot be accounted for by a simple communication/computation model. For example, a processor will likely spend time synchronizing with other processors, waiting for neighbor processors for data communication, and waiting for system delays. System overheads are associated with specifics of the hardware and Legion infrastructure such as the time required to resolve the physical location of a data object needed by the application. The overhead for PMHD3D can be estimated by:

$$Over_i = 16 - 1.5 * probSize/1000 + 0.094P^2$$

where P is the number of processors involved in the computation and $probSize$ is the height of the PMHD3D mesh.

$Over_i$ was estimated empirically using data from 106 individual application executions with problem sizes varying from 1000 to 6000 and with resource set sizes varying between 4 and 26. To determine the effect of the number of processors on overhead runs, runs were grouped by problem size and the corresponding execution times plotted against number of processors. For each set of runs performed with the same problem size, a quadratic fit was performed on the difference between the actual execution time and the predicted execution time (without the overhead factor). The quadratic factor varied between 0.090 and 0.096 with a mean of 0.094 (standard deviation of 0.0022). To determine the effect of problem size on overhead we used the same runs but did a linear datafit on the predicted/actual execution time difference with problem size.

3.2. Resource Selection

Resource selection is the process of selecting a set of target resources (processors in this case) that will be performance-efficient. Finding the optimal set of resources requires comparing all possible schedules on all possible subsets of the resource pool - clearly an inefficient process as the resource pool becomes large. Instead, we create several ordered lists of resources by employing a heuristic to sort candidate resources in terms of several definitions of *resource desirability*. Resource desirability is based on how resource characteristics such as computational speed and network connectivity will affect the performance of PMHD3D.

The resource selection process begins by querying Legion to discover the available set of resources. Effective evaluation of the desirability of each resource requires application-specific performance information as well as dynamic resource performance information. As of this writing, Legion collection objects report available resources and their static configurations but do not provide up-to-date dynamic information on availability, load, or connectivity. Accordingly, the list of available resources reported by Legion is used to query NWS for dynamic forecasts of resource availability, CPU load, and free memory for each host and of latency and bandwidth between all pairs of hosts. To obtain the computational cost per unit of the PMHD3D grid on each type of resource we used the benchmarking method described in Section 3.1.

Once the available resource lists and the dynamic system characteristics are collected, the list can be ordered in terms of desirability. We use three definitions of desirability of a resource: desirability based on connectivity, desirability based on computational power, and desirability based equally on the two characteristics. *Connectivity* is approximated by computing the latency and bandwidth between the resource in question and all other resources in the resource pool: as a metric we calculate the amount of time (seconds) it would take for the resource in question to exchange a packet of size 1 byte to and from every other host. *Computational power* is measured by the time (seconds) it would take the host to compute 1 point for 1 iteration based on the NWS predictions and the benchmarks we discussed earlier. The *balanced* strategy orders the resources based on an average of computational power and connectivity.

The resource set is sorted into 3 resource lists using the 3 notions of resource desirability. We then create subsets of the lists by selecting the n most desirable hosts from each list where $n = 2...maxP$ and n is even. We select multiple subsets from each list because it is often impossible to know the optimal number of hosts a priori. Once the subsets have been created the resulting group of proposed resource sets are passed on to the schedule creation step described in the next section. Although the approach described here is not guaranteed to find the optimal resource set, the methodology provides a scalable and performance-efficient approach to resource selection.

3.3. Schedule Creation and Selection

For each of the proposed resource sets, a schedule is developed. Essentially, schedule development on a given resource set for PMHD3D reduces to finding a work allocation that provides good time balancing. As in Section 3.1 work allocation is represented by x_i and is the height of the slab given to processor P_i.

One of the most important characteristics for any solution to this problem is time balancing: all processors should finish at the same time. Using the notation from Section 3.1,

$T_i = T_{i+1},\ i \in \{1 \ldots (n-1)\}$ and, since all of the work must be allocated, we also have $\sum_i x_i = probSize$. Taken together we have n equations in n unknowns and the problem can be solved with a basic linear solver. This approach was successful for the Jacobi-2D AppLeS [2] but is not powerful enough to incorporate several additional constraints required to develop good schedules for PMHD3D.

One of the important constraints for PMHD3D performance is the amount of memory available for the application. There is a limit to the size of problem that can be placed on a machine because if the computation spills out of memory, performance can drop by two orders of magnitude. To quantify this constraint a benchmark for application memory usage must be obtained by observing memory usage for varying problem sizes on each type of resource. Formally, this constraint becomes:

$$BMmem_i * x_i < MemAvail_i$$

where $MemAvail_i$ is the available memory for processor i (provided by the NWS memory sensor) and $BMmem_i$ is the memory benchmark (megabytes/unit) recorded for processor i's architecture.

We formalize the work allocation constraints as a *Linear Programming* problem (from now on simply LP), solvable with the simplex method [6]. In short, LP solves the problem of finding an extreme (maximum or minimum) of a function $f(x_1, x_2, \ldots, x_n)$ where the unknowns have to satisfy a set of constraints $g(x_1, x_2, \ldots, x_n) \geq b$ and both the *objective function* and the constraints are linear. The simplex is a well-known method used to solve LP problems. The simplex formulation requires that constraints are expressed in *standard form*; that is the constraints must be expressed as equalities and each variable is assigned a *non-negativity* sign restriction. There is a simple procedure that can be used to transform LP problems into a standard form equivalent.

We modified the time balancing equations to provide some flexibility for the constraints specification: expected execution time for any processor in the computation must fall within a small percentage of the expected total running time. This flexibility is beneficial, especially as additional constraints such as memory limits are incorporated into the problem formulation. The constraints are initially very rigid but can be relaxed in cases where no solution can be found given the initial constraints. The time balancing equations and the application memory requirements form the application constraints on which the simplex has to operate. The simplex formulation also requires specification of an *objective function* where the goal of the solver is to maximize the objective function while satisfying the simplex constraints. We use $\sum_i x_i$ as the objective function and search for a solution where all work is allocated.

For each of the proposed resource sets the simplex is used to create the best schedule possible for that resource set. We use a library [16] which provides a fast and easy to use implementation of the simplex. There are several benefits of using linear programming and the simplex method to create a good schedule:

- Linear programming is well known and commonly used so that fast and reliable algorithms are readily available.
- Once the constraints are formalized as a linear programming problem, adding additional constraints is trivial. For example, the FORTRAN compiler used to compile PMHD3D enforced a limit on the maximum size of arrays, therefore limiting the maximum units of work that could be allocated to any processor. This constraint was easily added to the problem formalization.
- The linear programming problem can be extended to give integer solutions, although the problem then becomes much more difficult. Currently the solver computes real values for work allocation and we redistribute the fractional work portions. In some problems a linear solution may be required for additional accuracy.
- In the case that a solution cannot be found, the simplex method provides important feedback. For this application, the simplex could not find a solution if the constraints were too restrictive. In this case the simplex is reiterated with successively relaxed constraints until a solution can be reached.

Once the proposed schedules are identified, schedule selection is surprisingly simple. The performance model is used to evaluate the expected execution time of each proposed schedule, and the schedule with the lowest estimated execution time is selected and implemented.

4. Results

The PMHD3D AppLeS has been implemented and we present results to investigate the usefulness of the methodology. The goals of these experiments were to:

- Evaluate the accuracy of our performance prediction model.
- Evaluate the ability of the PMHD3D AppLeS to promote application performance in a multi-user Legion environment.

The previous sections stressed the importance of the performance model for effective scheduling. In Section 4.2 we explain in detail results demonstrating the accuracy of the performance model. In Section 4.3 we present evidence that the scheduling methodology and implementation are effective in practice. Before discussing these results we first outline our experimental design.

4.1. Experimental Design

To evaluate the PMHD3D AppLeS, we conducted experiments on the University of Virginia Centurion Cluster, a large cluster of machines maintained by the Legion team (see [4] for more information on the cluster). The Centurion Cluster is continuously upgraded for new Legion version releases; during the 3-month period of the experiments, we used Legion versions 1.5 through 1.6.1. The cluster itself is composed of 128 Alphas and 128 Dual-Pentium II PCs; 12 fast Ethernet switches and a gigaswitch connect the whole cluster. Although we employed both Alphas and Pentiums during the development and initial testing process, we had multiple difficulties with Alpha Linux kernel instabilities and a faulty network driver which made our data for the Alphas machines unreliable. The results presented here are based only on the 400 MHz Dual Pentium II machines. We didn't employ the second processor on the Dual Pentium: therefore when we talk about host or machine we consider the machines to be uniprocessors. It is worth noting that many users only use one processor per node so that even a computationally intensive user will not affect CPU availability as much as might be expected. However, the two processors on each Dual Pentium machine utilize the same memory, sometimes leading to performance degradation due to overloaded memory systems. Inclusion of memory constraints in the performance model helped the AppLeS scheduler avoid overloaded memory systems.

We restricted our experiments to 34 machines for practical reasons: the dynamic information collected from NWS includes a large amount of data, even for a relatively small cluster. Limiting the resource pool did not impact investigations of application performance or schedule efficiency because, as will become clear, the parallelism available in PMHD3D for the problem sizes studied here is well below the 34 machine limit. As explained in Section 2.5 we used an IBP server running at all times at UCSD, while AppLeS acted as an IBP client retrieving the forecasts. This setup allowed us to obtain updated predictions for a large number of resources in a reasonable amount of time. On average it took less than 4 seconds to retrieve the data, with a minimum of 2.5 seconds and a maximum of 8.5 seconds.

To test the performance of PMHD3D under a variety of conditions, experiments were typically performed with maximum resource set sizes (from now on called *resource pool* or simply *pool*) of $4, 6...26$ and problem sizes of $1000, 2000...6000$. Problem size is the height of the data grid used by PMHD3D. The *pool* is the maximum number of machines the scheduler is allowed to employ. We test varying pool sizes to simulate conditions under which a user may be limited to a certain number of resources by cost or access considerations. Although our overall resource pool contains 34 machines in total, the maximum pool size we simulate is only 26. This choice was practical: we frequently found unavailable or inaccessible machines in our overall resource pool and so were never able to access all 34 machines at one time. Note also that the scheduler may determine that utilizing the entire pool is not the most performance efficient choice. In this case the pool is larger than the number of *target resources*.

The experiments presented in Section 4.2 were conducted under unloaded conditions while those presented in Section 4.3 were conducted under loaded conditions. The ambient load present during most of our loaded runs consisted of heavy use of some machines and light use of others. In order to investigate application performance we report performance results based on application execution time. However, there is a cost associated with using AppLeS to develop a schedule. We analyzed 43 runs in detail and the dominant scheduling cost is associated with querying the Legion Collection and the Legion context space. The time required to access NWS and IBP is on average less than 4 seconds. Once the system and performance information has been collected, the AppLeS required on average roughly 1 second to order the resources, create schedules, and select the best schedule.

4.2. Performance Model Validation

The performance model is the basis for determining a good work allocation and, more importantly, provides the basis for selecting a final schedule among those that have been considered. We tested model accuracy for a variety of problem sizes and target resource sets (see Figure 3). For the 62 runs shown in this figure the model accurately predicts execution time within *1.5%*, on average. The performance model consistently achieved this level of accuracy for other runs taken under similar conditions. Notice that as the problem size becomes larger, the smallest pool that we test also increases (i.e. the smallest pool for a problem size of 2000 is of size 4 while for a problem size of 6000 it is 12). This experimental setup was required by a limit in the g77 FORTRAN compiler we employed: no more than 507 work units could be allocated to any one processor during the computation.

Figure 3 demonstrates the importance of selecting an appropriate number of target resources for PMHD3D. For example, for a problem size of 1000 the minimal execution time is achieved when the application is run on 10 processors. If fewer processors are used, the amount of work per processor is high and the overall execution time is higher.

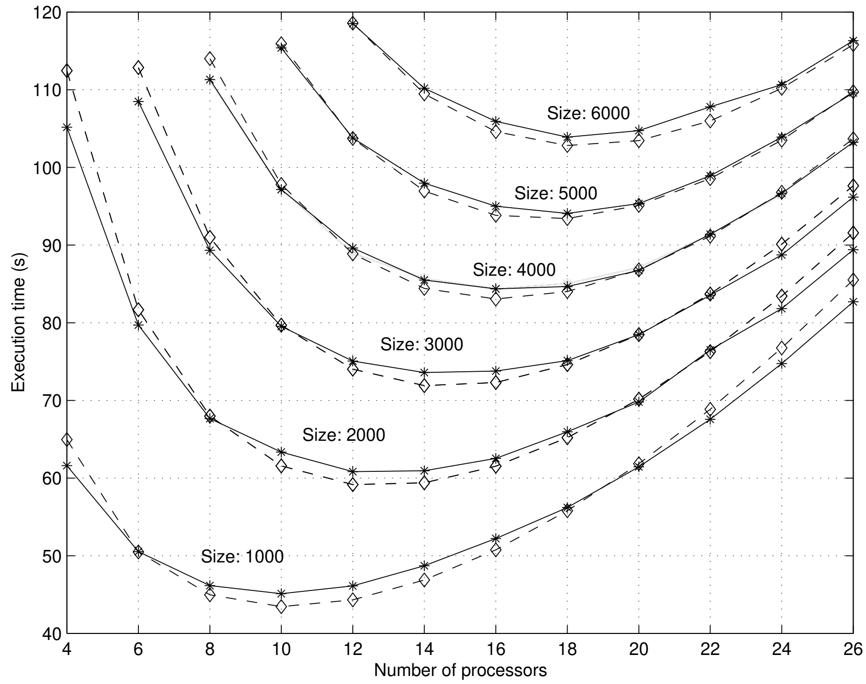

Figure 3. Model predictions (*dashed lines*) and observed execution time (*solid lines*) for a variety of problem sizes and pool sizes.

Table 1. Number of resources to target for various problem sizes under unloaded conditions. Optimal is the best choice, range indicates close to optimal choices.

$Size$	1000	2000	3000	4000	5000	6000
$Hosts$	10	12	14	16	18	18
$Range$	8-12	12-14	14-16	14-18	16-18	18-20

If more processors are used, the added communication and system overheads cannot be offset by the advantage of the additional computational power. Significantly, the performance model accurately tracks the *knee* (i.e. inflection point) in the curve and is thus capable of predicting the correct number of target resources, at least under these conditions. We report the optimal number of target resources for all problem sizes tested in Table 1. As will be obvious in Section 4.3, the optimal number of processors may vary with resource performance and dynamic system conditions as well as with problem size.

Figure 4 demonstrates the scheduling advantage of accurately predicting the correct number of processors to target. In these experiments the PMHD3D AppLeS was allowed to select any number of processors up to the maximum pool size. The PMHD3D AppLeS selects the maximum number of resources for each resource pool up to and including a size of 18. For resource pools of size 20 and larger the optimal number of hosts is 18 and the PMHD3D AppLeS correctly selects only 18 hosts.

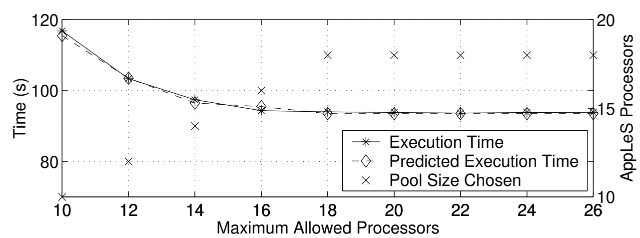

Figure 4. PMHD3D AppLeS predicted and actual execution times for a problem size of 5000.

4.3. Performance Results

Once we verified that the performance model is accurate in a predictable environment (i.e. where resources are dedicated), we turned our attention to considering the performance of the AppLeS in a more dynamic, unpredictable, multi-user environment. We begin by investigating the ability of PMHD3D AppLeS to compare available resources and select *desirable* hosts (computationally fast, well-connected, or both). To provide a comparison point we test the performance of another available scheduler, namely the default Legion scheduler. We conducted experiments in *runs*, namely back-to-back PMHD3D executions using the same resource pool and the same problem size but utilizing the PMHD3D AppLeS scheduler first and the default Legion scheduler second.

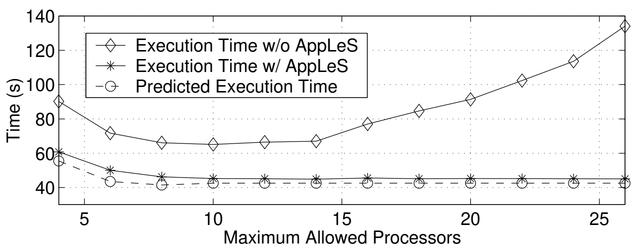

Figure 5. PMHD3D performance attained with and without the AppLeS scheduler for a problem size of 1000.

In Fig. 5 we show a series of runs comparing the two schedulers for a problem size of 1000. Clearly, the PMHD3D AppLeS provides a performance advantage for all resource set sizes tested. However, it is notable that the two execution time curves follow the same trend only when the resource pool is in the range of 4-12 hosts. When more resources are added to the pool the execution time achieved with the PMHD3D AppLeS remains constant while the default Legion scheduler execution time diverges. The default Legion scheduler allocates *all* available resources, a less than optimal strategy for PMHD3D. In Table 2 we report the typical number of processors selected by AppLeS for different problem sizes and resource set sizes.

For pool sizes of $4-12$ performance achieved via the PMHD3D AppLeS is consistently $20-25$ seconds lower than that achieved via the default scheduler. In this range of pool sizes, the PMHD3D AppLeS selects the maximum number of hosts available and so uses the *same number* of resources as the default Legion scheduler. The performance advantage is achieved by selecting "desirable" resources, i.e. resources that are computationally fast and/or well-connected. Figure 6 illustrates the load of all available machines just before scheduling occurred for the 18-processor run shown in Figure 5. Clearly, the PMHD3D AppLeS selects lightly loaded hosts (i.e. those hosts with high availability) while the default scheduler selects several loaded hosts. It is the load on these selected machines that causes

Table 2. Hosts chosen by PMHD3D AppLeS. The Legion default scheduler always selects the maximum number of hosts.

	Problem Size				
$Max\ Hosts$	1000	2000	4000	5000	6000
4	4	4			
6	6	6			
8	8	8	8		
10	10	10	10	10	
12	10	12	12	12	12
14	10	12	14	14	14
16	10	12	14	16	16
18	10	12	16	16	18
20	10	12	14	18	20
22	10	12	14	18	20
24	10	14	14	18	18
26	10	14	14	18	18

a performance disadvantage for the default scheduler. In a more heterogeneous network environment the connectivity of the hosts would also play an important role in host selection and resulting performance.

We obtained 83 runs comparing the default Legion scheduler to the PMHD3D AppLeS for a variety of problem sizes (1000-6000) and pool sizes (4-26). Figure 7 shows a histogram of the percent improvement the PMHD3D AppLeS achieved over the default Legion scheduler for the 83 runs (the average improvement was 30%).

Note that in a few runs there was little or no advantage to using the PMHD3D AppLeS. In these cases the processors were essentially idle and the pool size was below the optimal number so that the schedulers selected the same number of processors. In one run the PMHD3D AppLeS-determined schedule was considerably slower than that determined by the default Legion scheduler. In this case the scheduler created a schedule based on incorrect system information: NWS forecasts of CPU availability were unable to a predict a sudden change in load on several machines and the resulting schedule was poorly load balanced.

The Legion default scheduler was designed to provide general scheduling services, not the specialized services we include in the PMHD3D AppLeS. It is therefore not surprising that the AppLeS is better able to promote application performance. In fact, the PMHD3D AppLeS could be developed as a Legion object for scheduling regular, iterative, data-parallel computations, and this is a focus of future work. Using the PMHD3D AppLeS and the Legion default scheduling strategy as extremes, we wanted to explore a third alternative for scheduling – that of what a "smart user" might do: In a typical user scenario for a cluster of machines a user will have access to a large number of machines and will typically do a back-of-the-envelope static

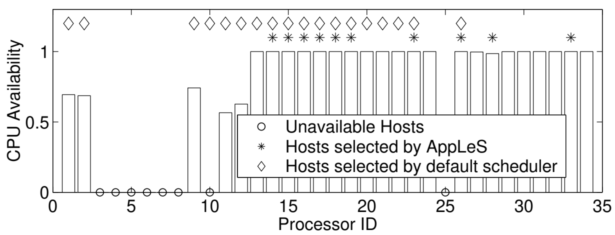

Figure 6. A snapshot of CPU availability taken during scheduling for the 18-processor run shown in Figure 5.

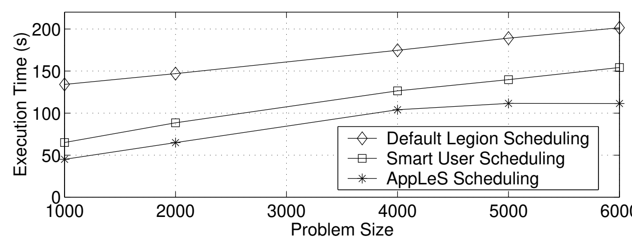

Figure 8. Performance obtained by three schedulers when each was given access to at most 26 processors.

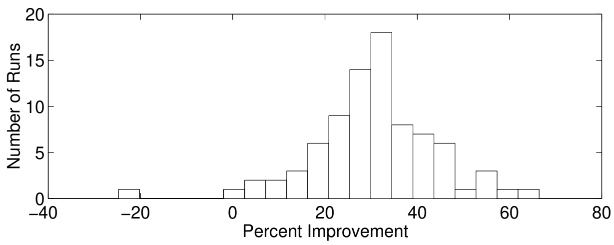

Figure 7. Range of performance improvement obtained by PMHD3D AppLeS.

calculation to determine an appropriate number of target resources given the granularity of the application. Although a user may correctly determine the number of hosts to target, accurate information on resource load and availability will be difficult or impossible to obtain and interpret prior to or at compile-time.

To simulate this user scenario, we developed a third scheduling method called the **smart user**. The *smart user* selects an appropriate number of hosts but does not select hosts based on desirability. Experiments were performed for problem sizes ranging from 1000 to 6000 with a pool size of 26 hosts. Figure 8 shows the performance obtained by the PMHD3D AppLeS, the default Legion scheduler, and that obtained by the *smart user*. In these experiments, the PMHD3D AppLeS provides a significant performance advantage over both alternatives.

5. Related Work

The PMHD3D AppLeS is an adaptation and extension of previous work targeting the structurally similar Jacobi-2D application ([2],[3]). Jacobi-2D is a data-parallel, stencil-based iterative code, as is PMHD3D. Both applications allow non-uniform work distribution, however Jacobi-2D employs strip decomposition (using strip widths) for its 2-dimensional grid while PMHD3D employs slab decomposition (using slab height) for its 3-dimensional grid. While the applications are structurally similar, PMHD3D required tighter constraints on memory availability and a more complex performance model. Additionally, PMHD3D was targeted for a much larger resource set (34 machines vs. 8). The availability of a larger resource pool for this work motivated the introduction of the quadratic overhead term in the PMHD3D performance model. Previous AppLeS work has not included the additional overhead of using extra machines in scheduling decisions.

As part of our previous work, we developed an AppLeS for Complib and the Mentat distributed programming environment. Complib implements a genetic sequencing algorithm for libraries of sequences. It is particularly difficult to schedule because of its highly data dependent execution profile. The implementation of Complib we chose was for Mentat [8] which is an early prototype of the Legion Grid software infrastructure. By combining a fixed initial distribution strategy (based on a combination of application characteristics and NWS forecasts) with a shared work-queue distribution strategy, the Complib AppLeS was able to achieve large performance improvements in different Grid settings [20]. In addition to AppLeS for Legion/Mentat applications, we have developed AppLeS for a variety of Grid infrastructures and applications [19, 21, 7].

In [10], the authors describe a scheduler targeting data parallel "stencil" applications that use the Mentat programming system. They specifically examine Gaussian elimination using a master/slave work-distribution methodology. While it is difficult to compare the performance of each system, their approach differs from AppLeS in that it requires more extensive modification of the application and it does not incorporate dynamic information.

6. New Directions

An ultimate goal is to offer the PMHD3D AppLeS agent within the Legion framework as a default scheduler for iterative, regular, stencil-based distributed applications. In particular, the scheduler's performance model is flexible enough to incorporate the requirements and constraints of other stencil applications and the characteristics of other platforms. To use this model for other appropriate applications, good predictions of megabytes transferred, number of messages initiated, overhead factor, benchmarks for program CPU and memory utilization over the different target architectures, as well as access to dynamic system information from NWS or a similar system would be required. Once obtained, these characteristics are used as inputs to the model without changing the model structure.

Portability and heterogeneity are also important. The AppLeS itself is written in C and Perl and has been compiled successfully and executed on various architectures and systems (Pentium, Alpha, Linux and Solaris). Initial results indicate that the scheduler can be used effectively on different target environments without changes to the structure of the performance model. For example, we used *mpich* on a local cluster for initial development and debugging. The schedule worked well with only the previously described changes in model input parameters.

Acknowledgements

The authors would like to express their gratitude to the reviewers for their comments and suggestions. The insight and focus provided by their comments improved the paper greatly. We thank the NWS team and Jim Hayes in particular for sharing their NWS expertise with us. We also thank the Legion team and Norman Francis Beekwilder in particular for sharing their Legion expertise with us.

References

[1] Application-Level Scheduling. `http://apples.ucsd.edu`.

[2] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proceedings of High-Performance Distributed Computing Conference*, 1996.

[3] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing*, 1996.

[4] Virginia Centurion Cluster. `http://legion.virginia.edu/centurion/facts`.

[5] S. J. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource management in Legion. In *Journal of Future Generation Computing Systems*, volume 15, 1999. page583-594 vol 15.

[6] D. Dantzig. Programming of interdependent activities, ii, mathematical model. *Activity Analysis of Production and Allocation*, July-October 1949.

[7] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, July 1998.

[8] A. Grimshaw. Easy-to-use object-oriented parallel programming with mentat. *IEEE Computer*, May 1993.

[9] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-area computing: Resource sharing on a large scale. In *IEEE Computer 32(5)*, volume 32(5), May 1999. page 29-37.

[10] A. Grimshaw, J. Weissman, E. West, and E. J. Loyot. Metasystems: an approach combining parallel processing and heterogeneous distributed computer systems. *Journal of Parallel and Distributed Computing*, June 1994.

[11] A. Grimshaw and W. Wulf. Legion–a view from 50,000 feet. In *Proceedings of High-Performance Distributed Computing Conference*, 1996.

[12] John Hawley. `http://www.astro.virginia.edu/~jh8h`.

[13] Internet Backplane Protocol. `http://www.cs.utk.edu/~elwasif/IBP`.

[14] Legion. `http://legion.virginia.edu`.

[15] G. Lindhal. Private communication.

[16] W. Naylor. wnlib. ftp://ftp.rahul.net/pub/spiketech/softlib/wnlib/wnlib.tar.Z.

[17] Network Weather Service. `http://nws.npaci.edu/`.

[18] ScaLAPACK. `http://www.netlib.org/scalapack/scalapack_home.html`.

[19] S. Smallen, W. Cirne, J. Frey, F. Berman, R. Wolski, M.-H. Su, C. Kesselman, S. Young, and M. Ellisman. Combining workstations and supercomputers to support grid applications: The parallel tomography experience. *Heterogeneous Computing Workshop*, May 2000. To appear.

[20] N. Spring and R. Wolski. Application level scheduling: Gene sequence library comparison. In *Proceedings of ACM International Conference on Supercomputing 1998*, July 1998.

[21] A. Su, F. Berman, R. Wolski, and M. M. Strout. Using AppLeS to schedule simple SARA on the computational grid. *International Journal of High Performance Computing Applications*, 13(3):253–262, 1999.

[22] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998.

Holly Dail is currently a M.S. student in the Department of Computer Science and Engineering at the University of California San Diego. She received a B.S. in Physics and a B.S. in Oceanography from the University of Washington in 1996. Her current research interests focus on achieving application performance in Computational Grid environments.

Graziano Obertelli is currently an Analyst Programmer in the Department of Computer Science and Engineering

at the University of California, San Diego. He received his Laurea in Computer Science at Univerità degli Studi, Milano.

Francine Berman is a Professor of Computer Science and Engineering at the University of California, San Diego. She is also a Senior Fellow at the San Diego Supercomputer Center, Fellow of the ACM, and founder of the Parallel Computation Laboratory at UCSD. Her research interests over the last two decades have focused on parallel and distributed computation, and in particular the areas of programming environments, tools, and models that support high-performance computing. She received her B.A. from the University of California, Los Angeles, her M.S. and Ph.D. from the University of Washington.

Rich Wolski is an Assistant Professor in the Department of Computer Science at the University Tennessee, Knoxville and a partner in the National Partnership for Advanced Computational Infrastructure. His research interests include parallel and distributed computing, on-line performance analysis techniques and software, compiler runtime system, and dynamic scheduling. He received his B.S. from the California Polytechnic University, San Luis Obispo and his M.S. and Ph.D. from the University of California at Davis/Livermore Campus.

Andrew S. Grimshaw is an Associate Professor of Computer Science and director of the Institute of Parallel Computation at the University of Virginia. His research interests include high-performance parallel computing, heterogeneous parallel computing, compilers for parallel systems, operating systems, and high-performance parallel I/O. He is the chief designer and architect of Mentat and Legion. Grimshaw received his M.S. and Ph.D. from the University of Illinois at Urbana-Champaign in 1986 and 1988 respectively.