EcoSystem: A Set of Grid Computing Tools for a Class of Economic Applications

Umakishore Ramachandran, Vladimir Urazov, Namgeun Jeong, Hasnain Mandviwala, David Hilley
College of Computing, Georgia Institute of Technology
{*rama*, *vurazov*, *namgeun*, *mandvi*, *davidhi*}@cc.gatech.edu

Daniel F. Waggoner, Tao Zha
Federal Reserve Bank of Atlanta
{*dwaggoner*, *tzha*}@frbatlanta.org

School of Computer Science Technical Report GT-CS-07-09.
Submitted for publication. Please do not distribute.

Abstract

Computational grids allow users to achieve high computational power with relatively simple resources. They are comparatively cheap and easy to set up, and pool the assets of potentially weak hardware together to create a superior resource. However, grids are generally not simple to use. Writing robust distributed applications can pose a challenge even to a seasoned programmer, but to a scientist whose expertise lies in another area, it could present a virtually insurmountable obstacle. To address this issue, we develop EcoSystem, a set of tools designed to make a grid more accessible to users without the intimate knowledge of distributed computing. The basic premise of EcoSystem is that the domain experts may not want to modify their sequential programs but would like to make use of the grid resources for running their applications in a painless manner. The toolset comprises a dynamic scheduler that allows exploitation of coarse-grained data, task, and pipeline parallelism that may exist in the applications; a scripting language that allows automatic generation of command line arguments for the executables; a monitoring component that allows the user to keep track of their execution; and a GUI client that runs on the users' desktops, for interacting with the toolset. The system has been in use by economists at the Federal Reserve Bank of Atlanta. In this paper, we present the architecture, implementation, and evaluation of EcoSystem.

1 Introduction

While the computational capacity of a computer continues to grow at an astounding rate, the computational requirement of some users still manages to outpace it, resulting in the need to pool resources into a grid of clustered machines. A cluster is simply a network of computers within a single administrative domain with potentially heterogeneous hardware configurations, but a common filesystem. A grid is a collection of clusters located across administrative domains that together provide even larger computational resources. Grid computing can therefore offer scientists the resources that would otherwise be unavailable to them on a single machine. Having numerous physical CPUs as computational resources makes grids well suited to coarse-grain parallelizable applications. This is why grid computing has been popular in the scientific community, which needs to process and analyze vast amount of data using a variety of algorithmic tools.

A prime example of an area where grid computing is useful is quantitative economics. In particular, Bayesian econometrics using Monte Carlo Markov Chain algorithms has become increasingly popular due to the growth in available computing power. Economists are able to create sophisticated models with the help of vast computational resources. Many of these applications are inherently parallelizable and would benefit greatly from being distributed over a grid of computers. However, while it is reasonably cheap and simple to set up a grid of low-cost hardware to produce high computational capacity, actually parallelizing applications on such a distributed platform is another matter altogether.

It is a non-trivial task to write robust and maintainable distributed applications, especially for domain experts who do not have the necessary skills in writing such programs. Additionally, even once a distributed application has been developed, the users of a grid have to perform numerous mundane mechanical tasks to actually execute the program. The grid needs to be accessed in a secure fashion. The users

need to handle the issues of uploading files to the grid, and remotely invoking their applications. This is the type of work that domain experts would rather avoid doing.

There is a significant body of work dealing with distributed programming models for grid computing, and general grid usability. Existing solutions range from low-level approaches that involve modifications to the source code of the distributed application, to high-level solutions that involve integration of distributed programming paradigms directly into the user's operating system (we conduct a more thorough examination of related work in Section 8). However, there is a lack of intermediate approaches that allow the users to leave their application virtually unmodified and provide them with tools to easily and securely utilize a grid to its fullest potential.

We present *EcoSystem*, a tool chain that makes grid computing more accessible to users who might lack the intimate knowledge of distributed programming and makes scheduling tasks on the grid more user-friendly. The basic premise of *EcoSystem* is that the domain experts may not want to modify their sequential programs but would like to avail of the grid resources for running their programs in a painless manner. Our development and deployment environment allows users to write a large application as a collection of small non-distributed programs, and launch them on the grid with ease.

EcoSystem is a complete system that is in use by economists at the Federal Reserve Bank of Atlanta for economic modeling applications. Through this work, we make the following contributions:

- *EcoSystem* provides a rich set of tools to interface the end users to the grid that includes:
  - A dynamic scheduler that allows exploitation of all the avenues of parallelism that may exist in long-running applications (such as coarse-grained data, task, and pipeline parallelism). The scheduler uses execution history of the individual program modules that make up the application to balance the load among computational resources.
  - A scripting language that allows the automatic generation of command line arguments for the program modules of the application that need to be run on the grid nodes.
  - A monitoring component that presents the current state of execution of the different program modules of the application to the user.
  - A GUI client that runs on the users' desktops and provides a simple and intuitive interface to the grid resources.
- We have carried out a preliminary evaluation of the system to quantify the overhead of scheduling and the potential benefits of the dynamic scheduler as compared to statically scheduling the resources harnessed from the grid.

2 Application Context

Grid computing offers a versatile solution to a multitude of computation-intensive applications. However, for a given application domain, there is potential for unique usability optimizations that provide a custom-tailored approach that is easy to use. In this paper, we focus on a class of simulation and modeling applications that involve long sequences of computation-intensive tasks executed with large unique data sets. In particular, to set the context for our work, we develop the architecture of *EcoSystem* with economic modeling applications in mind. However, we believe our tool chain generalizes to other application domains with similar computational requirements.

The class of applications we investigate exhibits parallelism of various types, each exploitable for increased application performance. Consider the problem of calculating the *Maximum Likelihood* (ML) estimate of a sophisticated model. This may require different optimization routines. For instance, one may begin with the unconstrained optimization routine and then use the converged result as a starting point for a constrained optimization routine. This process can continue with other optimization routines or can be re-run using the same sequence of optimization routines. An overall convergence criterion may be used to decide when to stop this process. The type of process where the output of one set of computations is used as input for another set is known as a *pipeline*. This optimization pipeline is summarized in Algorithm 1.

Algorithm 1 An example of pipeline parallelism.

repeat
  Optimization 1 $\rightarrow$ Optimization 2 $\rightarrow$ ... $\rightarrow$ Optimization n
until overall convergence is satisfied

Each optimization routine may take a non-trivial amount of compute time. The maximum likelihood estimate obtained this way typically delivers a more accurate numerical solution than using a simple optimization. While searching for the global maximum likelihood, this process will generally encounter multiple local peaks. These are used in the following step of the model computation and are thus also kept track of.

Given a maximum likelihood estimate, and a set of local peaks of the likelihood, produced in Algorithm 1, one may generate posterior samples of the model parameters using the values themselves or points in their neighborhoods. These samples can then be dynamically weighed and used

for computing the *marginal likelihood* of the model. This leads to a second pipeline of computational tasks, which can be visualized by Algorithm 2.

Algorithm 2 Another example of pipeline parallelism.

for each likelihood peak **do**
  peak → simulate posterior sample
end for
posterior samples → compute the marginal likelihood.

Again, each step of the pipeline is potentially computationally expensive.

The pipeline characterization of the economic modeling application computes a single model using one set of input parameters and data. However, we also encounter a *data parallel* scenario (fig. 1) that computes the same task pipeline using a different set of parameters and data. For example, one might wish to study the data at different granularity levels (quarterly data vs. monthly data), or to concentrate on specific time periods and/or economic regions. Alternatively, the data used by the first part of a modeling computation may be disaggregated (for instance, the economic data could be separated by the income levels of the types of individual families studied), and then the results may be collected and used in aggregate at the next step of the model. Either way, each pipeline described in Algorithm 1 is independent from the others, and hence, they can all be executed in parallel given available computational resources.

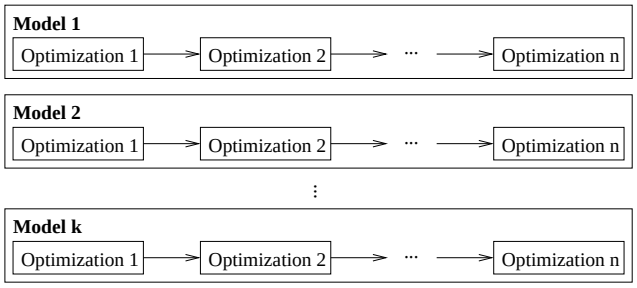

Figure 1. An example of data parallelism.

As mentioned above, each optimization step in computing the maximum likelihood estimate of a model is computationally expensive. Therefore, there is an opportunity to extract finer-grain parallelism from within these coarse-grain tasks. For example, the frequently used modified harmonic means approach computes an accurate estimate of the marginal likelihood of a complex model with the help of two additional steps.

In the first step, a large sample of the weight density function, the likelihood function, and the prior density function are generated by simulation using the posterior model distribution. The minimum of the posterior kernels is then obtained from this sample. We then compute a table of effective sample sizes for a set of lower bounds (greater than this minimum) and a list of scales of the weight density function covariance matrix. All table elements here can be computed in parallel.

In the second step, we compute the probabilities of lower bound restricted regions by simulation using the weight density distribution with different scales. Once again, the region probability calculation can be parallelized. A sample of marginal likelihood estimates and their standard error can then be calculated [28].

We say that a process where individual iterations are independent exhibits *iterative parallelism*, and the example described above is visualized by Algorithm 3.

Algorithm 3 An example of iterative parallelism.

for $i = 1$ to M **do**
  for $j = 1$ to N **do**
    (i^{th} lower bound, j^{th} scale) → computing the probability of the region restricted by the lower bound.
  end for
end for

Continuing with the example above, note that the *likelihood function* or *posterior probability density function* can be complicated and may contain local peaks. Searching for the global peak can be done in two steps. In the first step, the optimization routine described by Algorithm 1 can be used on a unique starting point. An arbitrary number of starting points (e.g. 500) is selected. This calculation can be parallelized with results sorted on disk for diagnostic analysis. In the second step, we begin with the point associated with the highest peak found in the first step. This point is perturbed in small and large steps to create an additional arbitrary number of points (e.g. 200 points). For each point, the optimization Algorithm 1 is used again. This step can then again be parallelized. We refer to this observation as *search parallelism*, and it is summarized in Algorithm 4.

Algorithm 4 An example of search parallelism.

for $i = 1$ to N **do**
  The i^{th} starting point → Algorithm 1
end for

Clearly, our application space is rich with parallelism along several dimensions. However, expressing and exploiting parallelism in a resource rich environment is a daunting task for domain experts. Fortunately, these types of parallelism are well known and studied in the high-performance computing community (consider a small sample of available work on parallelism [9, 16, 25, 7]). In the rest of the paper, we will detail how we can develop a collaborative

approach to help domain experts run such parallelism-rich applications on computational grids.

3 Problem Definition

The focus of this paper is to investigate how computer scientists may enable domain experts working on a specific class of applications to overcome the challenges of distributed programming on a grid. A domain expert is a scientist, an economist for example, who possesses extensive knowledge of the computational problem he is trying to solve. This domain expert possesses enough programming skills to implement the model, and enough knowledge about the model to identify opportunities for parallelization. However, he is not necessarily able to produce an efficient, robust and maintainable parallel application that would otherwise require technical knowledge in distributed computing. The domain expert may not know specific parallelization APIs such as *PThreads*, *OpenMP* or be aware of common parallel programming pitfalls such as race conditions, deadlocks, data contention, etc. A computer scientist, on the other hand, possesses intimate knowledge of tools used to parallelize an application, but does not necessarily have the domain knowledge to identify opportunities for parallelization without compromising application correctness. It would require the computer scientist to first spend a significant amount of time studying the underlying domain to achieve the desired goal. It is therefore the computer scientist's job to enable the domain expert to easily utilize the computational capabilities of the grid to the fullest extent without having to face the technical challenges posed by distributed computing.

Another aspect of a distributed target platform is data transfer from one task to another across machines. Domain expert may not be comfortable with network programming nor are they likely to be familiar with distributed shared memory libraries. We would like to provide the domain expert with a convenient way to share data between the different parts of the application.

Monitoring is also important in the context of distributed application development. It is an often the case that a bug may cause a program to silently terminate, or hang without an error log. This could potentially be caused by other tasks on remote nodes that terminated unexpectedly or encountered some other errors. We would like to enable domain expert to monitor the progress of the distributed application and be notified if certain execution anomalies such as task termination or loss of network connectivity are detected.

Next is the issue of configuration. Fundamentally, each program executes on the grid via a command that specifies its input parameters values that may represent special constants or file names containing input data. The same application then could also be re-executed for a range of different possible parameter values (data-parallelism). Enumerating all parameter values by hand is not only tedious, but also error-prone. We would like to enable the domain expert to produce such execution lists in a more automated and verifiable manner.

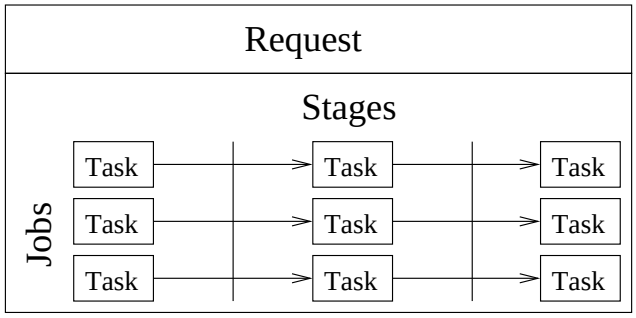

Figure 2. Visualization of the terms associated with units of work.

Finally, we deal with the issue of task distribution and task scheduling, which is also of crucial importance for efficient parallel execution. Distribution deals with the mapping of tasks to a computational resource for execution, whereas scheduling deals with the particular execution order for tasks assigned to a given resource. A good distribution policy is important to keep the load balanced among all available computational resources and a good scheduling policy helps avoid problems such as data-dependencies between tasks. We therefore enable an automatic and efficient distribution and scheduling policy that relieves the domain expert from having to deal with such chores.

In order to preserve flexibility in the configuration of the *EcoSystem* tool chain, we have developed a simple execution model conceptualizing the problem definition. The model is visualized in fig. 2.

Work is submitted by the user to the grid as a **request**. A request encapsulates all the work that needs to be done to solve a given problem. Requests internally consist of **jobs**. A job is a single line of execution of a pipeline from beginning to end. A single job takes in a single set of input data and configuration parameters, and outputs a single, fully processed, set of output data. Jobs are executed in **stages**. Each job has the same number of stages. The execution of a request starts with every job being ready to enter the first stage, and terminates when all jobs have finished the last stage. Collecting input data from storage, calculating the maximum likelihood estimate of a model, and formatting the output data are three examples of stages. A **task** is a single stage of a single job, as opposed to the stage being a collection of tasks for all jobs. A task is the smallest schedulable unit of work. It is a single execution of a single command, and is the building block of the bigger execution

framework.

4 Requirements

The domain expert faces numerous challenges when attempting to execute a model on a grid of computers in an efficient manner. To alleviate these concerns, a usability toolset can automate the repetitive technical tasks to various degrees. Solutions offered to a domain expert may vary anywhere from a low level communication substrate to high-level, domain-specific tools capable of deep automation. Obviously, there is tension between the flexibility that a system offers and the learning curve that it requires.

The approach presented in this study is an intermediate solution. We opt against a solution that requires modifications to the code of the original sequential programs developed by the domain experts, as we subscribe to the belief that it should not be the domain experts' responsibility to be aware of the details of distributed computing. We have also decided against tight integration with the operating system, and have chosen to provide a platform-independent graphical user interface client that, while providing the convenience of visualization, still leaves the user with great flexibility in terms of parallelization configuration. In other words, we would like to automate as much as possible the mundane mechanics of task execution on the grid, such as the decision which task to execute on which node, while leaving entirely open to the user the "creative" part of parallelization, that is, decisions like which tasks can be run in parallel, and which need to be performed sequentially.

Specifically, when using *EcoSystem*, the domain expert is responsible for the following part of parallelization:

- *Structuring the solution* as a set of coupled programs, while keeping in mind the possibility of exploiting various types of parallelism.
- *Understanding granularity of parallelism* at the task level. In other words, it is left for the user to decide just what a unit of work comprises. This involves intimate knowledge of the application and its computational and data requirements – the kind of knowledge only the domain expert can possess.

The computer scientist is responsible for handling the mechanics of distributing the workload as per the domain expert's requirements, through *EcoSystem*'s implementation:

- *Enabling structural representation* of requests. This calls for an intuitive way by which the domain expert can express the structure of the requests. With advances in *Graphical User Interface (GUI)* design, it makes sense that the domain expert should be able to represent the task dependencies graphically.
- *Enabling parameterization* of tasks. Each task in the representation may take on command line arguments for execution. Similarly, they may use data from files and produce data that goes into files. The system should absorb the responsibility for naming the parameters and the intermediate files automatically based on the coupling present in the representation.
- *Enabling mapping* of tasks to the computational resources. The structural representation simply specifies the coupling and the ordering of the execution of the tasks. Inherent in this coupling is communication and triggering of the task that is dependent on the completion of the previous task. The system should absorb this responsibility so that the domain expert has no need to know about the mechanics of the actual data communication or the synchronization that is inherent in the representation.
- *Enabling scheduling* of tasks on the computational resources. At any point of time there may be several tasks in the representation that are ready to execute. The system is responsible for scheduling these tasks on the available grid resources to increase the throughput and turnaround time for the application. The primary goal is efficient use of the computational resources offered by a grid, both by the automation tools the computer scientist provides, and in the way the tasks are scheduled on the grid.
- *Enabling Monitoring.* The user may wish to monitor the progress of the execution of a request. The system should facilitate this in a manner that is meaningful to the user.

5 Architecture

In order to address the issues detailed in the above sections, we have developed the *EcoSystem* chain of tools. Let us first examine the high level middleware architecture of the system. *EcoSystem* consists of four major components (fig. 3):

- *Scheduler* runs on the grid machines and distributes the computation tasks between the individual nodes.
- *Scripting Component* is employed by the users of the system to configure the tasks to be run on the grid.
- *Graphical User Interface Client* (*GUI Client*) runs on the users' desktops and is used to provide better visualization of the process of submitting a computation request to the grid.

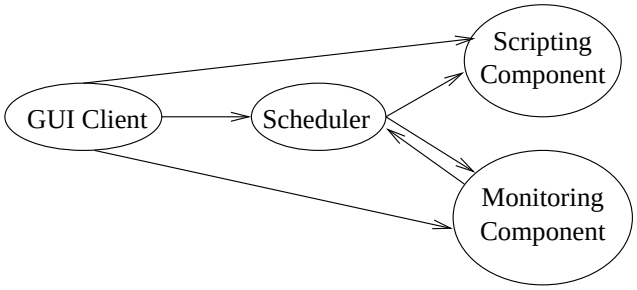

Figure 3. The high level architecture of *EcoSystem*. The GUI client facilitates the creation of scheduler configuration, employs the scripting component to validate the configuration, invokes the scheduler on the grid remotely, and communicates with the monitoring component to display the status of the currently running tasks. The scheduler parses the configuration using the scripting component, distributes the tasks among the grid nodes, and communicates with the monitoring component to provide it with updates on the status of task execution.

- *Monitoring Component* can be used to view both the status of the grid itself, such as the current load on the nodes, as well as the status of the individual tasks submitted by the users.

In the rest of this section, we will examine the finer details of the architecture of *EcoSystem*.

5.1 Scheduler

The *scheduler* is one of the fundamental parts of the *EcoSystem* set of tools. This component runs on the grid machines, and its main responsibility is to provide an efficient distribution of computation tasks between the nodes. One of the basic assumptions about the usage of our system is that it will be utilized in extremely compute-intensive conditions. To that end, there are two fundamental requirements that the scheduler must fulfill: it must consume few resources itself, so as to not introduce excessive overhead; and it must schedule computation tasks in such a way as to best utilize the resources the grid has to offer. In order to make smart decisions about the appropriate scheduling order, the tool takes into consideration the following information:

- *Application requirements* – the number of jobs and stages in the computation request.
- *Resource availability* – the number of nodes available to the scheduler for running computation tasks.
- *Execution history* – the data on how long it takes to execute individual tasks. This is used to ensure that the execution of every job is moving along smoothly.

5.2 Scripting Component

In addition to making efficient use of available computation resources, one of the other fundamental requirements we outline above is the ease of configuration. If the tool chain requires more configuration than manually executing the tasks on the grid, then it loses a large part of its appeal to the end users. Therefore, an essential part of *EcoSystem* is a scripting language that allows for simple and intuitive descriptions of the tasks that need to be executed on the grid and their specific ordering.

Every task is a standalone application written by the domain expert. Therefore, we represent each task by a command, exactly as one would invoke a program from a console command line (such as a Unix shell). This allows the scheduler to execute tasks on the grid by utilizing a simple system command function call, and thus remain decoupled from the specifics of the tasks it is managing. This also allows for the executables that implement the tasks to remain portable – since their implementations are in no way tied to the scheduler, they can be run manually as regular standalone applications. This also frees the end users from the burden of learning new libraries or interfaces – they simply implement the models as regular standalone programs, and merely need to configure the scheduler through our scripting mechanism in order to make the applications distributed.

As a result of the decision to make a single "system command" the smallest schedulable unit of work, we have also avoided having to subject the users of the system to custom configuration formats for their applications, and have deferred the specifics to them. This also means that the main mechanism to configure a single task is through command line parameters. This is of course a standard mechanism provided on the Unix operating systems, whereby every whitespace-separated string after the executable path and name is passed on to the executable's entry point as a string value.

Naturally, we could do without a scripting component, and opt to simply provide the scheduler with a list of commands to execute, perhaps numbered in some fashion so as to allow for dependency resolution. However, that would go against our "ease of configuration" principle, outlined above, as long command lists are not easily readable and modifiable (see Section 6.2 for a more thorough discussion on the merits of our scripting language vs. other approaches).

The scripting component of the tool chain is utilized mainly by the scheduler. The scheduler's configuration essentially enumerates the files that contain the scripts describing the tasks to be scheduled on the grid in a concise and readable fashion. The scheduler then processes the script files with a script translator that generates the specific

list of commands that the scheduler needs to run.

5.3 Graphical User Interface Client

The end user interacts with the grid using a GUI client that serves two main purposes: (a) it allows the user to configyre the tasks to be executed on the grid (i.e. create the coupled stages representation shown in fig. 2; and (b) it allows the user to visualize the progress of the current requests launched on the grid. It simplifies and expedites the user experience for several reasons. First, it provides an environment for the user to create and validate the task configuration, utilizing the scripting component. Second, the client can display the status of the grid to the user, allowing either for manual selection of the nodes to run the computation tasks on, or for automatic selection of the appropriate nodes based on their current load. Third, the client simplifies the submission of the requests to the grid, by automatically uploading the scripting configuration files to the nodes, and remotely invoking the scheduler, as appropriate. Finally, the GUI provides a way for the user to monitor the requests currently managed by the scheduler, and view their specific status, such as the number of tasks already completed, the average time per task in a stage, etc.

5.4 Monitoring Component

The monitoring component is a service running on the grid that provides information about the status of the nodes and currently running jobs.

The high level grid information includes items like the current and historic CPU and memory usage per node, network utilization, remaining disk capacity, etc. This type of data can be used by grid administrators to monitor the overall health of the entire system, as well as that of the individual nodes. It can also be useful for the end users of the application, because they can to some extent self-regulate their grid usage. Consider the situation where multiple users gravitate towards a small subset of nodes. This could happen, for example, due to the hardware characteristics of these nodes, such as a 64-bit architecture, as opposed to a 32-bit architecture on the other machines. In this case, these few favored nodes might become overloaded, while the other ones remain idle and waste capacity. The system-wide information that the monitoring component provides can help to easily identify this problem, and attempt to remedy it, whether by attempting to restructure the users' applications to be more tolerant of hardware differences, or, failing that, to simply encourage them to coordinate their grid usage times between each other to more evenly distribute utilization throughout the day.

The monitoring component also provides detailed information on the requests currently being executed on the grid.

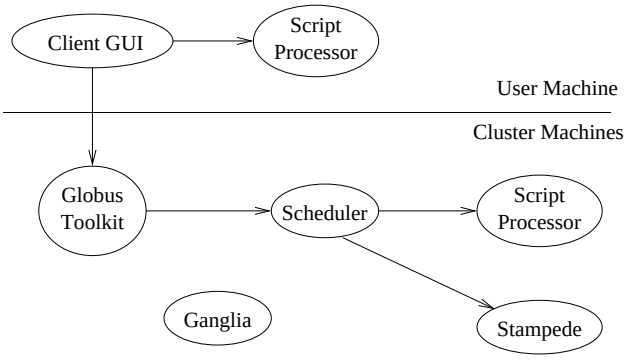

Figure 4. Components of *EcoSystem* implementation.

It is able to report the number of requests running, and the IDs of the users who initiated them, as well as the exact status of each task on a per-request basis. This is directly useful for the end users of *EcoSystem*, because it allows them to closely follow the execution of their tasks. The users can be notified of failures in the execution of any of the tasks, which helps identify issues with their applications or their task configuration. Users can also monitor individual stages for run times, to detect performance deficiencies of various types, and deadlock or infinite loop related bugs.

6 Implementation

In this section, we discuss the specifics of the implementation of the architecture described in Section 5. Some of the components described in the architecture section are implemented with multiple tools or services in practice, and relations between the different parts of the implementation are visualized in fig. 4. Note that our implementation currently runs on a single cluster of machines; however, we plan to generalize it to extend across administrative boundaries and support entire grids.

6.1 Scheduler

The scheduler component of *EcoSystem* is implemented in *C*. The main reasons behind this choice are performance and portability considerations. This choice has also enabled us to utilize the functionality of the *Stampede* [24] distributed programming library. The use of this library greatly simplifies the implementation of the scheduler. Stampede allows us to concentrate on the actual scheduling logic, as it provides significant functionality for dealing with the common issues that a distributed programming environment presents.

First, Stampede automatically instantiates the distributed application on the additional nodes of the grid, once it is

started on one of them, and this process is highly configurable.

Second, Stampede provides functionality for memory objects to be distributed and shared between the instances of the application running on different nodes. This easy distribution of data allows us to decentralize the making of scheduling decisions. Each node knows which tasks have been completed, how long the tasks take on average, etc. This allows an instance of the scheduler running on a node to make the decision about the next task locally, without directly consulting the master node for this information. Additionally, because Stampede provides synchronization constructs for access to distributed shared queues, it is easy to ensure that multiple nodes do not attempt to execute the same user level task simultaneously.

The use of Stampede and decentralized decision making allows us to ensure that the scheduler itself runs efficiently. But of course, the main purpose of the scheduler is to distribute the computation tasks between the nodes of the cluster in such a fashion that the resources offered by the system are used to the greatest potential. To that end, we have developed a customized scheduling algorithm. We have designed this algorithm with the following principles in mind:

- Each job should progress through the stages as fast as possible.
- We should not have certain jobs remaining idle for extended periods of time, i.e. all jobs should be moving along.
- The jobs that are far behind of the average progress should be given more resources than those that are far ahead, since we assume that the user wants to see all the results together, rather than individually.

To achieve these goals, the scheduler keeps track of certain data on a per-stage basis that is accessible to all the nodes executing a request. We have a global Stampede queue containing the current progress of each job, as well as a *scoreboard* which is a distributed shared object, containing, for each stage, the following information:

- *jobs_ready* – the number of jobs that are pending for execution at this stage.
- *jobs_running* – the number of jobs that are currently executing in this stage.
- *jobs_finished* – the number of jobs that have progressed past this stage.
- *execution_time* – the total amount of time spent at this stage for those jobs that have already been finished.

Using the above data, the scheduling algorithm runs in three passes, with a distinct policy at each pass refining the decision. First, the scheduler picks the latest stage in the request that has tasks ready to be executed in it, as described in Algorithm 5. This essentially ensures that we attempt to move each job through the stages as fast as possible, and is a basic *First In/First Out* algorithm.

Algorithm 5 Policy 1 of the scheduling algorithm – return the index of a stage to process in FIFO order.

```
for i = stage_count − 1 to 0 do
  if stages[i].jobs_ready ≥ 1 then
    return i
  end if
end for{If we haven't returned anything after the loop, there are no pending jobs at the moment.}
```

Once the scheduler has gone through the first policy, it knows whether there are tasks pending, and if so, what is the latest stage with a task ready to be executed. At this point, the application goes to the second policy in the scheduling decision process, which is described in Algorithm 6. This policy ensures that if there are pending tasks in a stage, at least one of them is being executed. This means that there are no "runaway" jobs that get finished before any other job is even started, and that at any given time, if there are enough nodes involved in the process, each job is being worked on.

Algorithm 6 Policy 2 of the scheduling algorithm – ensure that every stage with pending tasks has at least one task being processed.

```
for i = stage_count − 1 to 0 do
  if stages[i].jobs_ready ≥ 1 and stages[i].jobs_running = 0 then
    return i
  end if
end for
```

Finally, if the second policy fails to find a suitable stage to execute, that is, if every stage with pending jobs is executing at least one, we proceed to the third scheduling policy. This policy also ensures that jobs do not fall too far behind and all of them are approximately at the same stage of progress. It assigns workers to stages, preferring those that historically have taken longer to complete and those that have the most tasks pending. The third scheduling policy is described in Algorithm 7.

In addition to the scheduling services, this component also provides detailed logging on the progress of task execution. This information is necessary for the monitoring component to produce reports on the individual requests the users make to the cluster.

Algorithm 7 Policy 3 of the scheduling algorithm – pick a stage that has the maximum execution time.

```
result ← 0
max_avg ← 0.0
for i = stage_count − 1 to 0 do
  if jobs_ready[i] ≥ 1 and jobs_finished[i] ≥ 1 then
    average_time ← execution_time[i] / jobs_finished[i]
    if max_avg ≤ average_time then
      max_avg ← average_time
      result ← i
    end if
  end if
end for
return result
```

6.2 Scripting Component

For configuring the tasks to be executed on the cluster, we have developed a custom scripting language. This solution balances two important aspects of configuration: brevity, and simplicity. On the one hand, it is preferable not to configure the scheduler with a plain list of commands for it to execute. A command list can become unmanageable quickly. For instance, in one modeling application in use at the Federal Reserve Bank of Atlanta (see Section 7), the users are interested in varying three parameters of a model call. These parameters have 2, 3, and 7 interesting values respectively, which amounts to a list of 42 commands. It is thus very easy to make a mistake when a parameter value needs to be added, removed or changed. On the other hand, using a pre-existing general-purpose scripting language certainly can make the configuration more brief, but would introduce unnecessary complexity and an additional learning curve for the users. Therefore, we have created a custom scripting language, the *S*tring *E*numeration *L*anguage (*SEL*), geared specifically at describing lists of commands, and automatically producing sets of command-line parameters. The language is very specific to our problem domain and has a minimal feature set. This reduces the learning time necessary to gain command of the language. We have developed a pre-processor for SEL in *C*, that automatically converts SEL scripts to plain command lists that can be used directly by the scheduler component.

SEL is based on the concept of *ordered string lists* (OSL). An OSL is simply a collection of strings, where their order is important. For example, the set of strings $\{\text{“}foo\text{”}, \text{“}bar\text{”}\}$ is considered as an OSL with the string “foo” on the top of the list and the string “bar” at the end of the list. We expand on the concept of OSL to develop SEL resulting in its powerful expression capabilities. In SEL, OSLs can be defined as follows:

$$\$varid = \{\text{“}foo\text{”}, \text{“}bar\text{”}\};$$

Here, $\$varid$ represents a variable name or identifier to an OSL. Again, an OSL can consist of one or more strings, a combination of number ranges and number lists, or even another $\$varid$ representing another pre-defined OSL. Examples of such expressions are:

$$\{1-5\}$$
$$\{1,2,3,4,5\}$$
$$\{1,2,3-5\}$$
$$\{\$OSL1, \$OSL2\}$$

With strings comes the need to concatenate them together. Concatenation is an operation whereby two or more strings are joined together to form one string. e.g. the string “foo” and “bar” can be concatenated to form the string “$foobar$”. This operator is extended to work on OSLs. For example, two OSLs can be concatenated to form a single OSL as follows:

$$\$OSL1 = \{\text{“}foo\text{”}, \text{“}bar\text{”}\};$$
$$\$OSL2 = \{\text{“}baz\text{”}, \text{“}bam\text{”}\};$$
$$\$result = \$OSL1\$OSL2;$$

Here the OSL $\$result$ is $\{\text{“}foobar\text{”}, \text{“}foobam\text{”}, \text{“}barbaz\text{”}, \text{“}barbam\text{”}\}$. Note how each string in $\$OSL1$ is concatenated in order with every string in $\$OSL2$. Clearly, a precedence order is visible here where a given string in $\$OSL1$ is concatenated with *every* string in $\$OSL2$ *first*, before continuing on the next string in $\$OSL1$. This implied default left-to-right precedence in operators such as concatenation can be changed by explicit specification of the precedence order using positive integers, where *increasing* values define *lower* precedence. e.g.:

$$\$OSL1 = \{\text{“}foo\text{”}, \text{“}bar\text{”}\};$$
$$\$OSL2 = \{\text{“}baz\text{”}, \text{“}bam\text{”}\};$$
$$\$result1 = \$OSL1(1)\$OSL(2);$$
$$\$result2 = \$OSL1(2)\$OSL(1);$$

The first concatenation $\$OSL1(1)\$OSL2(2) \rightarrow \{\text{“}foobaz\text{”}, \text{“}foobam\text{”}, \text{“}barbaz\text{”}, \text{“}barbam\text{”}\}$ illustrates the default concatenation order in SEL. The second concatenation $\$OSL1(2)\$OSL2(1) \rightarrow \{\text{“}foobaz\text{”}, \text{“}barbaz\text{”}, \text{“}foobaz\text{”}, \text{“}barbam\text{”}\}$ inverts the concatenation precedence resulting in an OSL with the same set of strings but listed in a different order. These simple expressions comprise of the simple yet poweful String Enumeration language.

6.3 Graphical User Interface Client

The primary functionality of the client is to provide the users with a central façade for all their needs as pertains to

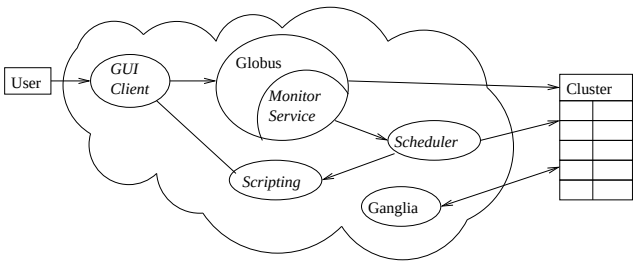

Figure 5. The user's perspective of the *EcoSystem* tool chain. The tools whose names are in italics have been implemented by the authors for this study, whereas the tools in regular font have been developed by third parties. The user performs all tasks through the GUI client. The client communicates with the cluster through the Globus Toolkit. Globus allocates resources and instantiates the scheduler as needed. The scheduler employs the scripting component to process the configuration files and executes tasks on the cluster. Ganglia performs cluster-wide monitoring duties, while the monitoring component running inside Globus produces request-specific reports.

creating, submitting, and monitoring computation requests (fig. 5). As such, it contains several pieces of functionality. First, the GUI provides an editing environment for request configuration files, allowing users to create and validate the scripts. Second, the client allows users to manually filter the nodes that the request will be running on, to account for possible hardware or other restrictions they might want to impose on the execution of their applications. Third, the GUI allows for a single-click request submission process. And finally, the tool provides monitoring capability for currently running jobs.

The client is implemented in *Java*, mainly for the purposes of portability and simple and maintainable creation of GUI components. The tool runs on the users' machines, simplifying the process of submitting computation requests to the cluster. Using the GUI client removes the need for the users to directly interact with the cluster nodes through other means, such as SSH access, or SCP to upload the request configuration files.

A sample screenshot of the GUI client is presented in fig. 6. This is the main window of the client, and is the one that the users will spend the most time in. On the left side of the window is a list of stages that the request is broken up into (in this case, there are five). When the request is submitted to the cluster, the stages will be executed in the same order as they are listed. Next, there are the UI control buttons, which allow the user to create, rearrange, validate and delete stages. Dominating the center of the window is the editor panel that allows the users to provide the content of a stage, i.e. the scripts that will be executed by the scheduler. To the right of the editor is a set of controls that allow

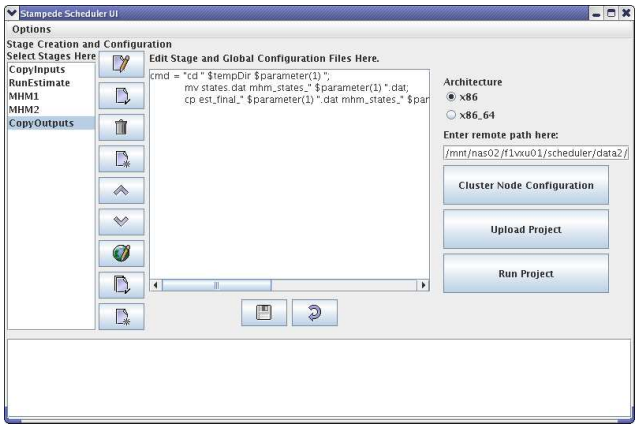

Figure 6. A sample screen of the GUI client.

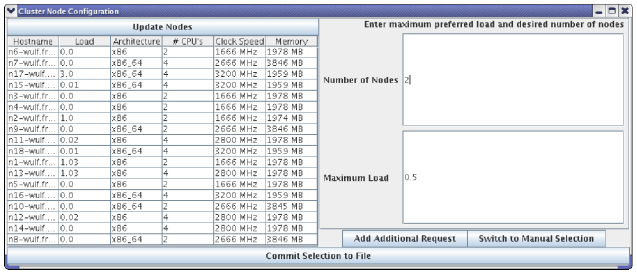

Figure 7. A GUI client form that allows the users to filter the nodes to execute the computation tasks on.

the user to configure where on the remote server the configuration files will reside, to select a subset of nodes to run the request on, and to submit the request to the cluster. On the bottom of the GUI is a status output panel that shows the output of the tools invoked by the client, such as the scripting component, remote job submission, etc..

While the main window contains the functionality that the users will be spending most of their time using, the GUI client contains some additional features. First, users can bring up a list of nodes available on the cluster (fig. 7) from the main window. This allows the users of the application to either automatically or manually filter the nodes that the computation request will be executed on. The choice to limit the nodes can be made automatically based on the desired architecture of the machine, as well as the current CPU utilization.

Another feature the GUI client provides is the ability to view the list of requests currently executing on the cluster (fig. 8). This includes the ID of the user that started the job, the configuration of the job, and the start date and time of the job.

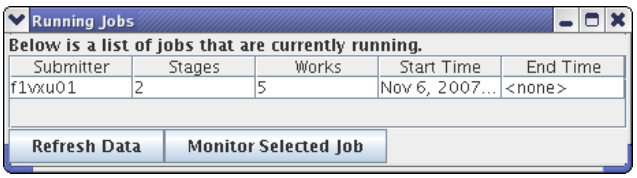

Figure 8. A GUI client window displaying a sample currently running job.

6.4 Globus Toolkit

The GUI client automates the process of uploading the files to the cluster and of invoking the scheduler, but this convenience for the end users comes at the price of initial setup. The *Globus Toolkit* [12] is used for performing these actions as it provides an efficient and secure way to submit requests to the cluster and to upload files to the nodes. Globus has been selected for its rich set of features, established user base, and extensive documentation, however it does add some extra initial setup and configuration time to the system.

In the *EcoSystem* tool chain, the Globus Toolkit is installed on the cluster, and acts as an intermediary between the GUI client and the scheduler running on the nodes. Globus provides several crucial components that make the implementation of the client substantially simpler, more robust, and more portable.

First, the Globus Toolkit provides an extensive security framework, which includes authorization components that can be integrated with an existing system-wide authentication setup running *Kerberos* and *LDAP*. This is a very important feature for us, because in our system, the cluster is accessible from outside the intranet to allow researchers offsite to utilize its computational power. This makes it necessary to utilize a robust authentication system, and Globus provides that functionality.

Second, Globus provides an interface to upload files to the nodes, and invoke commands on the nodes remotely. This allows us to almost completely remove the need for the users to directly access the cluster machines, as they can use the GUI client on their own computers for the vast majority of tasks they need to run. Of course, it is still necessary for the users to ensure that the applications they are planning to run on the cluster, as well as the input data files the models rely on, are accessible from the nodes. That, however, can be accomplished without directly accessing the cluster nodes. In our setup, for example, each user has access to a personal storage folder that is automatically shared through an *NFS* mount between all the client machines and the cluster nodes.

Third, the Globus Toolkit's primary client API's are provided in Java, so it is a perfect fit for our purposes, as it removes the need to provide any sort of wrappers for the application, and keeps it portable, so it can still run on all architectures that can run Java and are supported by Globus.

6.5 Monitoring Component

The monitoring component of the *EcoSystem* tool chain performs two major tasks: reporting cluster-wide information about the nodes, and request-specific information about the tasks currently running on the cluster. To that end, it is divided into two major parts.

The first part of the monitoring component is responsible for reporting information about the overall health of the cluster. For this purpose, we utilize the *Ganglia* [26, 22]. Ganglia is a lightweight scalable system to monitor the state of individual nodes in a large cluster via a graphical interface. State variables such as machine load, network traffic, and memory usage can be monitored constantly to give a consolidated view of multiple state parameters of the cluster. Ganglia has three main components, the web front-end, the data-gathering server, and the data-gathering clients. The data-gathering clients reside on the individual computers and are responsible for reading the state of the physical machines and delivering the information to central servers. The data-gathering servers collect the state information from clients and store them in a time-sequenced database. The front-end, which is accessible via any standard web browser, is a dynamic PHP website that generates graphs on-the-fly to display a historic overview of the cluster state.

The second part of the monitoring system is responsible for reporting the progress of the requests currently being executed on the cluster. This is actually not any individual tool, but is rather achieved through cooperation between the other components of *EcoSystem*. The first step in the reporting process is provided by the scheduler, which creates detailed logs about the execution of the tasks it schedules. Essentially, all the information that the tool keeps to make scheduling decisions, it writes into logs. The second link in the reporting chain is a service that runs within Globus. This service is notified by Globus when the scheduler is remotely invoked, and when it finishes execution. The service continually monitors the logs produced by the scheduler, records the data and processes it for easy reporting. The client application is able to query the data recorded by the Globus service, and present it to the user in the form of a report. The report lists the requests currently managed by the scheduler on the cluster, which user started them, and how far along they are.

7 Performance Evaluation

The two main goals of the system presented above are usability for the domain experts and potential performance gains. Therefore, we present an evaluation of the system with these goals in mind, and demonstrate that *EcoSystem* fulfills both. The following sections detail our measurements and findings. In addition to tests artificially created specifically to exercise *EcoSystem*, we give the system a real-world context by taking an existing economic modeling application in use at the Federal Reserve Bank (FRB) of Atlanta, adapting it for use with *EcoSystem* and presenting a comparison of the new setup with the original process.

7.1 Usability

There are several aspects to the usability of a cluster. One is the convenience of use. *EcoSystem* provides a GUI client for the users to launch applications on the cluster, whereas the process previously in use at FRB Atlanta involves command-line access. While the convenience of a GUI against that of a command line interface is somewhat subjective and is a matter of personal taste and experience, the developments in user interface design suggest that graphical interfaces are easier to grasp for users without a strong computer science background.

Another, more measurable, aspect of the usability of a cluster is the amount of configuration and setup that goes into launching a distributed application. To assess this, we perform a case study on an application that is currently in use by economists at FRB Atlanta, and compare the size of the configuration and script files used without any specific cluster usability tools with that for *EcoSystem*.

The cluster currently in use for these applications is comprised of 18 nodes (a heterogeneous collection of Intel Pentium processors with different CPU ratings) running the *Fedora Core 5* operating system. In order to distribute model computations on the nodes without any special tools, two shell scripts are in use. The first resides on each cluster node and is responsible for copying the data into the machine, running the computations necessary, and copying the results back to shared storage. This script is 31 lines (869 bytes) long, and generally remains unchanged between runs of the model, as it takes all the information it needs as a set of command line parameters. This is the script whose logic our scheduler configuration captures. The second script in use is the actual coordinator of work. It contains all the permutations of inputs that the users would like to analyze and calls the first script remotely on the compute nodes of the cluster. This script has roughly 42 lines (2913 bytes). It needs to change constantly – every time the set of inputs changes. The second script also has a hard-coded assignment of computational tasks to cluster nodes, and hence needs to be modified if the users' preference for nodes changes for some reason (for example, hardware preference, current load on the nodes, etc.). This script does precisely the type of error prone work that we would like to abstract away from the users and automate as much as possible.

We have created a set of configuration files to perform the same work utilizing the *EcoSystem* tool chain. Our configuration can also be logically broken up into two parts. First, there are the "stage" configuration files, that is files that specify the types of commands we would like to execute at each stage for a job (e.g., copy the input files in, run the models, collect and report the results). This part of the configuration fits into 7 lines (615 bytes), a value roughly comparable to the size of the original "logic" script. This is to be expected, and shows that the introduction of a custom scripting language did not make the expression of task logic any more cumbersome.

The second part of configuration is a file that specifies exactly which data needs to be fed into the models, the parameters of the models to run, etc. This is the dynamic part of the configuration, the one that needs to change essentially with every run, and the one that introduces the most potential for errors in the original set of scripts. This part of configuration for *EcoSystem* fits in a mere 7 lines (288 bytes) – an improvement of 83% in lines (or 90% in bytes) over the original script. This is where the introduction of the custom scripting language described in Section 6.2 really shines. The users do not need to enumerate every possible combination of input parameter values that they are interested in. Instead, they simply list the input data files and the parameter values that they desire to study, and the combinations are then automatically produced internally in the scheduling process. This allows for a far more readable configuration file and reduces the potential for human error.

7.2 Performance

We have evaluated the performance of the scheduler on two levels: how well the scheduling algorithms are able to balance the load, improving upon a per-determined hard coded assignment of computational tasks to nodes of the cluster, and how much overhead the scheduling application adds to the execution of the computational tasks. For these performance tests we have run our experiments on the same cluster of Linux machines. The cluster is comprised of heterogeneous hardware, but for consistency, we limited ourselves to the computers having four 2.8GHz 32-bit CPUs and 2GB of RAM.

We have also conducted controlled experiments, where our work is simply executing "sleep" statements of varying length. This allows us to measure the performance of the scheduler, since it provides a consistent workload, whose running time we know with reasonable precision, and which

Scheduler Overhead	Space Overhead	Startup Time	Time Per Task
	46 MB on each node	$\hat{2}$ sec	0.1 sec
Synthetic Benchmarks	Speedup vs. static scheduling		
	Config. 1	Config. 2	Config. 3
	24.8%	33.1%	8.9%
Application Performance	Speedup vs. static scheduling		
	FIFO Only	All Policies	
	28%	41%	

Figure 9. Summary of all performance evaluations.

does not introduce significant additional memory usage to the scheduler's footprint.

See fig. 9 for a summary of our performance findings.

Scheduler Overhead. To measure the performance of the scheduler itself, we have run several experiments with a static job length and a varying number of jobs and stages. As a result, we have found that the scheduler introduces a roughly 46MB memory overhead on each cluster node, and takes on average 0.1 of a second per scheduling decision, plus about 2 seconds of startup time for a reasonably sized task configuration. Considering that the application deals with inherently compute-intensive, long running jobs, this is reasonably negligible. In the case study we have performed on the economic modeling application, a single request would typically be comprised of about 40 jobs with 5 stages each. The running time of each task normally varies with the desired precision of the models' outputs. While at the lowest possible precision a single model call can be finished in as little as 20 seconds, at the regularly used precision levels, a single call can take up to 8 hours or longer to compute.

Synthetic Benchmarks. Now that we are comfortable that the scheduler does not introduce an unreasonable performance penalty, we would like to test the performance of our scheduling algorithm against a pre-determined static assignment of computational tasks to cluster nodes. Clearly, the amount of speedup this gives to the user depends largely on the nature of work that is being done on the cluster, so we examine several different scenarios.

First of all, we should note that if there are at least as many processors available as jobs that need to be run, then regardless of the scheduling decisions we make, the best execution time we could hope for is constrained by the longest running job. Therefore, we concentrate on scenarios where the number of jobs exceeds the number of available processors. Here, one of the advantages of *EcoSystem* is that each

	Stages			
Jobs	3	3	3	3
	1	1	1	1
	1	1	1	1
	1	1	1	1

Figure 10. Task running times in minutes for a sample setup with a single job dominating the others.

	Stages			
Jobs	3	3	3	3
	1	1	1	1
	3	3	3	3
	1	1	1	1

Figure 11. Task running times in minutes for a sample setup with a two jobs dominating the other two.

job is broken up into tasks by stage, which allows a finer scheduling granularity.

The first scenario we examine is a commonly occuring one, where a single job takes substantially longer than the other ones, as illustrated in Figure 10. In this case, a static assignment of computational tasks to nodes is clearly inferior if we do not have prior knowledge of how long each job will take. In this test, we have four jobs with four stages, and only two processors to run the jobs. A static assignment without prior knowledge of job lengths might assign two jobs to each of the processors, which would take a total time of 16 minutes to complete all jobs. In *EcoSystem*, since a processor attempts to find a task to execute the moment it becomes idle, one of the processors works on the first job, and in the mean time, the other processor finishs the other three jobs, thus reducing the total running time to 12 minutes – an improvement of 25%, and the theoretical best in this case.

The problem of static scheduling can become even worse if, as shown in Figure 11, we have two jobs that run longer than the other two. In this case, the user might get lucky and make the best possible running time of 16 minutes, but might also get unlucky and produce the worst running time of 24 minutes. In this case, our scheduler will always finish in the best running time.

Of course, this type of performance could be guaranteed by a scheduler that assigns computations to nodes on a per-job basis. Our scheduler, however has a higher granularity and can reassign computations on a per-task basis. We use the following experiment to demonstrate the gains of performance from this feature. Suppose a single task (that is, one stage of one job) dominates the running time of the request,

	Stages			
Jobs	1	4	1	1
	1	1	1	1
	1	1	1	1
	1	1	1	1

Figure 12. Task running times in minutes for a sample setup with one stage dominating the others.

as shown in Figure 12. In this case, the best a job-level scheduler can hope to do is complete all computation in 11 minutes. *EcoSystem*'s scheduler, on the other hand, can take advantage of higher granularity of scheduling, and does in fact make sure that all jobs are roughly as far along into the processing as the other ones, hence it completes these computations in 10 minutes – a speedup of 9%.

Application Performance. In addition to performing these contrived, and very controlled experiments to show the soundness of our scheduling algorithms, we also examine the gains in performance for the FRB Atlanta research application. As discussed above, the running time of a single model call depends largely on the desired output precision. To make the experiments more managable and repeatable, we turn the precision down from the commonly used levels, so that each model call runs on the order of minutes, instead of hours. In order to keep the majority of the cluster free for everyday use, we also limit the experiments to 15 jobs with 5 stages each, being serviced by 10 processors. Additionally, for each experimental setup, we have performed 10 runs, so as to average out possible inconsistencies due to factors beyond our control. The results of the experiments are as follows. Running this set of computations using the original process completes, on average, in 864 seconds. Performing this work, utilizing *EcoSystem* with only the FIFO scheduling enabled takes 621 seconds – a 28% run time speedup. This is to be expected, though of course the exact number depends largely on the specific static mapping – one could always attempt to tweak it to make the entire set of jobs run faster. Running these computations with all three of our scheduling policies enabled, produces results in a total run time of 503 seconds, on average. This is a further improvement of 19%, showing that the extra scheduling policies really do offer an additional performance advantage. Again, the specific number is determined by the exact jobs being run in this case. The performance gains from the additional scheduling policies go down if all the tasks in a request have very similar execution times.

The set of experiments shown above demonstrates some of the more measurable advantages offered to cluster users by *EcoSystem*. First, it provides a more concise and expressive way to configure the computation tasks. Second, *EcoSystem* provides an automatic dynamic scheduling system that offers users significant speedups in total running time of their applications.

8 Related Work

EcoSystem touches many aspects of parallel and distributed application development including programming interfaces, the representation of parallelism, communication, scheduling, execution monitoring, etc. Naturally, there is a huge body of relevant prior work addressing one or more of these issues, ranging from very low-level to very high-level. Since our work is HPC-oriented and utilizes a Grid-computing framework, we will focus primarily on solutions popular in those areas. PVM [8] and MPI [11] provide relatively low-level message-oriented communication layers and some collective communication operations for general distributed programming. MPI has been extended from a cluster-centric model to Grid computing domains in implementations like GridMPI [17], MPICH-G2 [19] and MGF [14].

Significant relevant work exists in distributed programming models for Grid computing [20, 30]. Like *EcoSystem*, these programming models and tools seek to simplify the process by which Grid applications are constructed, but their approaches vary significantly – some simply provide communication primitives for explicit distribution, while others provide language-centric or domain-specific higher-level support for constructing Grid applications. The aforementioned Grid-based MPI efforts and GridRPC [27] provide explicit communications primitives, while systems like ProActive [3] implement Grid-enabled Java distributed objects. Alchemi [21] and GTPE [29] provide Grid-wide distributed thread abstractions.

Most of the preceding systems are extensions of standard distributed programming paradigms to Grid. *EcoSystem*'s approach is more closely related to Nimrod/G [5], a set of higher-level runtime components for "parameter sweep" style applications. GRID Superscalar [2] is also closer to *EcoSystem*'s approach, because it provides explicit support for modeling parallel applications as higher level dependency graphs, handling dependency management, scheduling and communication. Unlike *EcoSystem*, GRID Superscalar uses a single program with special annotations to infer input/output dependencies and task decomposition. A stub compiler is used to generate a decomposed application suitable for distributed execution on the Grid.

PBS [15], Condor/Condor-G [13], Maui [18], IBM's LoadLeveler, and Sun Grid Engine [23] all provide batch-oriented schedulers and resource management solutions. Nimrod/G [5] and the the AppLeS Parameter Sweep Tem-

plate [6] both provide scheduling mechanisms targeted towards parameter sweep applications. The broader AppLeS project [4] is an effort to design scheduler "templates" appropriate for different classes of applications. Gridbus Broker [31] extends Nimrod/G's model to support external data dependencies and more runtime dynamism in application parameters, and Nimrod/G's original model has also been extended to support sequential/staged dependencies [1]. *EcoSystem*'s scheduler is targeted for a specific type of parameter sweep scheduling including sequential dependencies, and uses execution history to ensure uniform forward progress in staged runs. *EcoSystem* leverages Stampede [24] for communication and spawning tasks.

For resource monitoring, *EcoSystem* uses Ganglia [26] because it is widely-used and well-integrated with Globus MDS [10]. Network Weather Service [32] also provides monitoring features and is used in AppLeS, but NWS is more oriented towards end-to-end network monitoring while Ganglia is more endpoint-oriented.

9 Conclusion

Grids offer users superior computing resources. However, to a domain expert with no intimate knowledge of distributed programming, a grid may prove difficult to use, and even more so – to utilize to full capacity. To address this issue, we have presented *EcoSystem*, a tool chain that automates certain mechanics of distributing an application. We have discussed *EcoSystem* in the context of a class of economic modeling applications, though we believe the concepts presented above generalize to other fields with similar requirements. We have shown a set of experiments to quantify some of the more measurable advantages offered to grid users by *EcoSystem*: simple configuration and efficient execution of tasks. We have also discussed some existing work related to grid usability and have demonstrated that *EcoSystem* contributes a novel combination of front-end and back-end tools to enhance the overall user experience.

While *EcoSystem* is a mature system that is currently in use at the Federal Reserve Bank of Atlanta, there remain areas for future development. First, the current implementation assumes a single administrative domain, and runs on a single cluster of machines. This limitation is purely implementation-specific, as the architecture makes no assumptions that might preclude the use of computational resources across administrative bounds. Therefore, one of our goals is to extend the implementation to be able to operate on an entire grid of machines. This work is simplified by the choice of the Globus Toolkit for interactions between the user desktops and tools of *EcoSystem* running on the grid, since Globus has been designed with grids in mind. Second, the current scheduling algorithms assume that there is a fixed number of jobs with a fixed number of stages running for any given request. We would like to generalize this to more complex dependency graphs for computation tasks. For example, certain stages may have more tasks than others. This could happen if a stage of model computations involves filtering out certain data categories, or aggregating data into fewer pieces for further processing. One can imagine even more complex data flow graphs with jobs merging and forking at multiple points. Our architecture is general enough to accommodate for this extension, and our current implementation is modular enough to allow this to be achieved via fairly localized changes.

10 Acknowledgements

The authors would like to thank Mark Jensen and Eric Wang, economists at FRB Atlanta, for their input on the design and evaluation of *EcoSystem*; Mary Hnatyk and Lane Blount, of FRB Atlanta, for their support of the project; as well as Gabriel Heim and Steven Dalton, students at the College of Computing at Georgia Institute of Technology, for their contributions to the development of *EcoSystem*.

References

[1] S. Ayyub, D. Abramson, C. Enticott, S. Garic, and J. Tan. Executing Large Parameter Sweep Applications on a Multi-VO Testbed. In *Proceedings of CCGRID '07*, pages 73–82, Washington, DC, USA, 2007. IEEE Computer Society.

[2] R. M. Badia et al. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, June 2003.

[3] L. Baduel et al. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[4] F. Berman and R. Wolski. The AppLeS Project: A Status Report. 8th NEC Research Symposium, Berlin, Germany, May 1997.

[5] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. *High Performance Computing (HPC) ASIA*, 2000.

[6] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of the Super Computing Conference (SC'2000)*, Nov 2000.

[7] M.-H. Chen, Q.-M. Shao, and J. G. Ibrahim. *Monte Carlo Methods in Bayesian Computation*. Springer-Verlag, New York, 2000.

[8] J. Dongarra, G. A. Geist, R. Manchek, and V. S. S. m. Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, 7(2):166–174, 1993.

[9] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[10] S. Fitzgerald. Grid Information Services for Distributed Resource Sharing. In *Proceedings of HPDC '01*, page 181, Washington, DC, USA, 2001. IEEE Computer Society.

[11] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[12] 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.

[13] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, 2002.

[14] F. Gregoretti et al. MGF: A grid-enabled MPI library. *Future Generation Computer Systems*, 24(2):158–165, 2008.

[15] R. L. Henderson. Job Scheduling Under the Portable Batch System. In *Proceedings of IPPS '95*, pages 279–294, London, UK, 1995. Springer-Verlag.

[16] W. D. Hillis and J. Guy L. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[17] Y. Ishikawa, M.Matsuda, T.Kudoh, H.Tezuka, and S.Sekiguchi. GridMPI The Design of a Latency-aware MPI Communication Library. In *SWoPP 03: Summer United Workshops on Parallel, Distributed and Cooperative Processing 2003*, August 2003.

[18] D. B. Jackson, Q. Snell, and M. J. Clement. Core Algorithms of the Maui Scheduler. In *Proceedings of JSSPP '01*, pages 87–102, London, UK, 2001. Springer-Verlag.

[19] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: a Grid-enabled implementation of the Message Passing Interface. *Journal Parallel Distributed Computing*, 63(5):551–563, 2003.

[20] C. Lee and D. Talia. *Grid Computing*, chapter 21 – Grid Programming Models: Current Tools, Issues and Directions, pages 555–578. Wiley Series in Communications Networking & Distributed Systems. John Wiley & Sons, Ltd., 2003.

[21] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. Alchemi: A .NET-Based Enterprise Grid Computing System. In *Proceedings of ICOMP'05*, Las Vegas, USA, June 2005.

[22] M. Massie, B. Chun, and D. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. In *Parallel Computing*, April 2004.

[23] W. G. S. Microsystems). Sun Grid Engine: Towards Creating a Compute Power Grid. In *Proceedings of CCGRID '01*, page 35, Washington, DC, USA, 2001. IEEE Computer Society.

[24] U. Ramachandran et al. Stampede: A Cluster Programming Middleware for Interactive Stream-Oriented Applications. *IEEE TPDS*, 14(11):1140–1154, 2003.

[25] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of MICRO 27*, pages 63–74, New York, NY, USA, 1994. ACM.

[26] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler. Wide Area Cluster Monitoring with Ganglia. In *Proceedings of Cluster Computing '03*, pages 289–298, December 2003.

[27] K. Seymour et al. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *Proceedings of GRID '02*, pages 274–278, London, UK, 2002. Springer-Verlag.

[28] C. A. Sims, D. F. Waggoner, and T. Zha. Methods for inference in large multiple-equation Markov-switching models. Working Paper 2006-22, Federal Reserve Bank of Atlanta, 2006.

[29] H. Soh et al. GTPE: A Thread Programming Environment for the Grid. In *Proceedings of ADCOM '05*, December 2005.

[30] H. Soh, S. Haque, W. Liao, and R. Buyya. *Advanced Parallel and Distributed Computing: Evaluation, Improvement and Practice*, volume 2 of *Distributed, Cluster and Grid Computing*, chapter 8 – Grid Programming Models and Environments, pages 141–173. Nova Science Publishers, 2006.

[31] S. Venugopal, R. Buyya, and L. Winton. A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids. In *Proceedings of MGC '04*, pages 75–80, New York, NY, USA, 2004. ACM.

[32] R. Wolski. Dynamically forecasting network performance using the network weather service. *Journal of Cluster Computing*, 1:119–132, January 1998.