54083

An Advanced User Interface Approach for Complex Parameter Study Process Specification in the Information Power Grid

Maurice Yarrow, Karen M. McCann, Rupak Biswas, Rob Van der Wijngaart

Computer Sciences Corporation, Mail Stop T27A-1
NASA Ames Research Center, Moffett Field, CA 94035
{yarrow,mccann,rbiswas,wijngaar}@nas.nasa.gov

Abstract

The creation of parameter study suites has recently become a more challenging problem as the parameter studies have now become multi-tiered and the computational environment has become a supercomputer grid. The parameter spaces are vast, the individual problem sizes are getting larger, and researchers are now seeking to combine several successive stages of parameterization and computation. Simultaneously, grid-based computing offers great resource opportunity but at the expense of great difficulty of use. We present an approach to this problem which stresses intuitive visual design tools for parameter study creation and complex process specification, and also offers programming-free access to grid-based supercomputer resources and process automation.

1 Motivation and Background

Only a decade ago, the solution of the partial differential equations required for the evaluation of aerospace vehicle flow-fields typically involved a single discretization zone and was performed on a single processor of a high-speed compute engine that was usually situated locally. These compute tasks were so costly in CPU cycles that the notion of performing parameter studies was usually ignored. Now, however, the flow-solvers are typically parallel codes, and the problems to be solved involve large numbers of interrelated discretization grids. The compute engines are frequently large parallel machines with multi-gigabyte memories and terrabyte disk farms. Researchers have available the resources not only of their own laboratories but also those of other laboratories and the shared resources at large computer centers which are accessible via fast networks. Parameter studies are now quite feasible and are being performed on a regular basis by many researchers who require solution information throughout a given aerospace vehicle flight regime. The difficulties now have shifted to the manual creation of these parameter studies and to the difficulties associated with launching and managing the large number of jobs required by these studies. Modern aerospace flow-solvers frequently require large sets of discretization grids which describe the geometry of the aerospace vehicle. Subsequently, these solvers produce as output large collections of date files. Up to now, most parameter studies were performed with 2-dimensional flow solvers, but researchers are now starting to use 3-dimensional solvers for their parameter studies.

Recent developments in grid-based "metacomputing" such as Globus [ref] and Legion [ref] have created opportunities for running parameter studies on remote networked high-performance compute servers which constitute a shared resource for participants. But these opportunities come at a price, and that price is the proliferation of job control language to support these capabilities. This has placed an onus on users of these "Information Power Grids" who are typically engineering and research code users but who are not generally well prepared or enthusiastic about learning or creating the requisite control language scripts for managing parameter studies on a metacomputing grid.

We briefly describe the notion of a "parameter study" by giving two general examples. Simulation codes produce results which are the solution to a scientific or engineering problem defined for some set of values ("parameters") which are the input to the code and which define the original problem. Varying these parameters typically produces

different results. Varying these parameters through some specific range (the "parameter space") will yield a set of results describing behaviour in this parameter space. It is this family of related results that are sought; this is called a "parameter study" (sometimes written as "parametric study"). As a second example of a parameter study, we point to Monte Carlo simulations. Monte Carlo code runs typically must be repeated some substantial number of times in order to be able to accumulate sufficient data to be statistically meaningful, i.e., the results must be accumulated into ensemble averages. This too, then, can be considered a type of parameter study where the parameter to be varied simply is the seed for random number generation. In this case, though, the parameter varied does not actually have any significance as a physical parameter of the problem.

The end product of creating and launching parameter studies is typically a large suite of result files which must be postprocessed and/or moved to some form of long-term storage. Furthermore, parameter study users must be able to keep track of these results and log into a scientific diary such particulars as nature of the solved problem, location of the result files, history for the individual runs, and any other associated information. Being able to easily recreate and then modify the parameter study is also an important need for many users.

As a result of these concerns, we undertook to determine if there existed a parameter study capability that fulfilled the need of users at our computer center. The only tools that seemed applicable for these tasks were the historically related Clustor and Nimrod codes [ref]. We tested and surveyed the capabilities of these tools. Both Clustor and Nimrod were able to generate and launch simple parameter studies. They also implemented an internal "meta-language" for describing parameter study creation. Additionally, they made it easy to parameterize command line arguments.

However, these tools did not fully meet our requirements nor the requirements of our users. Some of these requested capabilities are as follows. Our users must have access to multiple job submission environments. These usage environments must include any combination of PBS, LSF, MPI, Globus, Condor, Legion. Also, users are currently requesting the ability to create what we call "multi-stage" parameter studies (we give a detailed example of this later in this article). Users also need a "fire-and-forget" capability, i.e., once the parameter study suite is created, it should be possible to initiate job launching and then shut down the parameter study tool entirely. We required that job submission should continue autonomously, and without the continued presence of the parameter study tool. Users are also requesting a fairly comprehensive level of job auditing and scientific diary capability, which we see as the secretarial side of a problem solving environment. On the development side, we needed to design a parameter study tool that could be easily extended using a very high level rapid-prototyping language (such as Perl). This is because we envisioned using the tool as a testbed for experiments in parameter study creation models, job submission models, and complex process specification models. We also needed to be able to use the tool to generate shell scripts designed for parameter study job submission and also for complex process job submission (visual scripting). It was essential that the script generation process be very flexible.

2 Definition of the Problem

Creating and launching parameter studies without the assistance of automating tools is laborious, tedious, and error-prone. By briefly examining the stages required for this task, we will be able to discern the nature of the inherent problems. The first task in this process is to create the set of parameterized input files which incorporate the sets of values representing the parameter study. These parameter value sets are a Cartesian product of the individual sets of values over which each of the parameters of interest vary. As such, the total number of combinations (the "parameter space") can very quickly get to be very large, and creating these sets of input files manually will be time-consuming and error-prone. Each of the resultant input files represents a user-program run. Launching these jobs involves setting up partitioned file spaces in which these jobs can be run, supplying each of the runs with all required input files, submitting the jobs, and then monitoring job progress and managing the output from these runs. We have adopted the policy that all of these functions must be automated and integrated into a single Graphical User Interface (GUI); this was our first design requirement. Our second design requirement was that of extreme simplicity of use. We believe that users are very sensitive to ease of use issues, and that users will ultimately avoid process automation tools that are difficult or non-obvious in use. The third design requirement is that a parameter study tool must be able to automatically self-document the actions it performs. If it cannot do this, users will very quickly be mired in a morass of old runs - hundreds, even thousands of runs whose origin and purpose is no longer obvious even to the originating user. This therefore forces a competent parameter study tool to be part "problem solving environment" and part scientific diary. The fourth design requirement is that of job submission flexibility in a scientific computation environment that is currently in flux. This is because "Grid" based computing has added new complexity and new layers of "job control

language" to the task of submitting jobs.

3 Parameterization of Program Input Files

In order to minimize the difficulty of building a parameterized input file set, we have created within the parameter study GUI an integrated, special-purpose text editor. This editor, in fact, has unusual editing capabilities; it functions exclusively to allow the *graphical* selection of the appropriate parameter data fields and to allow the user to designate the set of values for each of the candidate parameterized fields. This parameterizer is depicted in Figure 1. The value

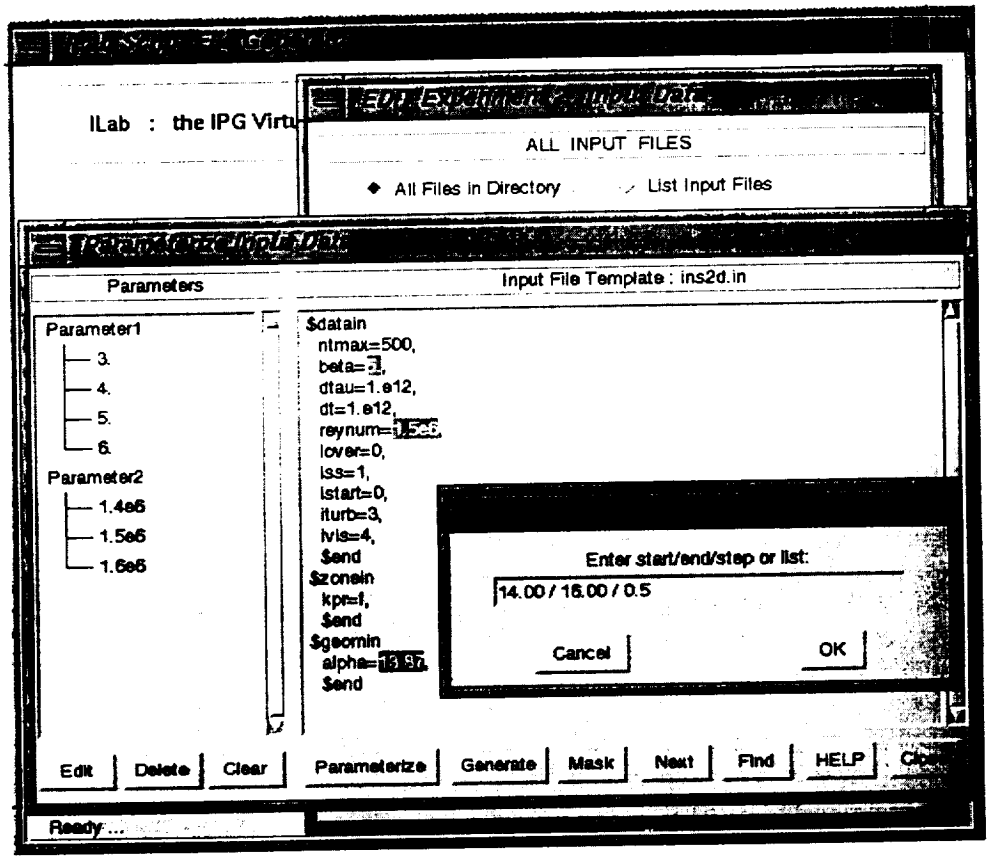

Figure 1: ILab parameterization screen

sets can be specified either as a list or by min/max/increment. During this process, the user first selects (highlights) with "mouse" those ASCII text fields within the input file which will be parameterized. In Figure 1, "beta" and "reynum" (known to ILab as Parameter1 and Parameter2) have already been parameterized; their value sets are displayed in the left window. Currently the user is specifying the third parameter in the "Set Param Values" dialog. If several fields must be parameterized in tandem (example: multiple occurrences of, say, "timestep" for each of several related grid files), these fields can be selected at this stage and parameterized in unison. After text selection of the appropriate fields, the user then enters list or range values for the selected fields. Lastly, the set of parameterized input files is generated. These files constitute an n-dimensional Cartesian product of the individual parameter sets. As a simple example, if three input values are to be parameterized (a 3-dimensional parameter space), the first with the set of values $\{1, 2, 3, 4\}$, the second with $\{hello, goodbye\}$, and the third with $\{3.14, 2.718, 1.618\}$, then the size of this Cartesian product set is $4 \times 2 \times 3 = 24$, so twenty-four parameterized files will be produced.

Because the file parameterizer is integrated within the parameter study GUI, and because its use is intuitive, the process of parameterization of the input files has thus been made quite trivial. Additionally, a "most-recently-used" capability saves the current parameterization state for future reference and for re-use or modification.

4 Job Masking Capability

One of the necessities of a parameter study program is to provide "masking" capability for a set of parameterized input files. Users have requested this ability since they know that certain parameter combinations will produce an unsuccessful run of the scientific program under consideration. Typically, they want to specify combinations of parameter values that will be excluded from the set of parameterized files and their associated script files. ILab's "Edit Parameters" screen - the special purpose editor mentioned above - has a pop-up dialog dedicated to this purpose. User can enter any number of "rules" for masking, and each rule must specify at least two parameter comparisons (but is not limited to two comparisons.) For example, if the user is varying Parameter1 from 1 to 10, and Parameter2 from 55 to 75, and wants to exclude those combinations where Parameter1 is greater than 9 and Parameter2 equals 60, the rule would be entered as:

```
Parameter1 > 9 && Parameter2 == 60
```

This syntax, which is the same in Perl, C, C++, and Java, was chosen since our users are likely to be familiar with it. The names "Parameter1" and "Parameter2" are assigned in order by ILab to the values being parameterized. (ILab, of course, has no way of knowing the actual names of parameters in the user's input files since there is no requirement that the input have labeled data. In the example in Figure 1, it just so happens that the user is parameterizing a Fortran "namelist" file with labeled fields, but ILab itself only requires that the input be ASCII.) By using Perl's "eval" function, we can easily interpret the above rule with a minimum of parsing, and use it to delete job objects from user's Experiment list.

5 Coding Model and Language Choice

We have chosen to construct our parameter study GUI using Perl5 and the Tk user interface construction tool kit. In addition, we have used the Perl generation capabilities of the "SpecTcl" Tk GUI generation IDE, a free software tool available from Sun Computer Corporation. Our choice of Perl5 was based on its strong character string manipulation and built-in regular expression capabilities, strong list and sortable associative-hashtable datatypes, and its simple-to-use object-oriented features. Also, Perl is relatively ubiquitous and is amongst the fastest interpreters commonly available today. Altogether, these features make Perl an excellent choice for true rapid-prototyping. Though we cannot exactly quantify the savings in the coding effort, we believe, based on prior experiences, that the equivalent functionality would require two to three times as much C++, or Java.

6 Object-Oriented Data Structures and Strategies in ILab

We used Perl "packages" (the equivalent of a "class" in C++ and Java) to hold all ILab data, both persistent and transient. Figure 2 depicts the data structures hierarchy. An `Experiment` package holds all persistent data: this data is serialized (written en mass retaining data structure hierarchies) to and from disk with the use of Perl's `Data::Dumper` module. In order to hold down the size of the `Experiment` package, several other packages apportion data that has to be held in lists or arrays, and `Experiment` has arrays of these packages: a `ParamFile` package for each input file to be parameterized, a `ParamData` package for each variable that is being parameterized in each input file, and a `Job` package to hold run-specific data. `ParamFile` and `ParamData` hold file and variable specific data while the Experiment is being created and edited. In order to run user's Experiment, a list of `Job` packages is created: some `ParamFile` and `ParamData` data is transferred to Job packages, and additional data is added. The organization of data in the `ParamFile` and the `ParamData` packages is "orthogonal" to the way the same data is organized in the array of Job packages: this simplifies script creation, submission, and monitoring. Essentially, during editing/creation, data is in arrays of arrays; during submission/monitoring, the same data is "flattened" out into a 1-D array of Job packages. Both sets of data are serialized when an Experiment package is serialized.

Each window or dialog box is also a package, and these packages hold transient data: user interface references and data as necessary, and also "mirror" portions of the current Experiment data. This duplication of data makes it easier and more robust to edit previously entered data, since user can change data, and then cancel the changes, without the necessity of having to restore the original data. Another important advantage is gained from the "mirror" and "orthogonal" approaches: the trade-off is more data, less code. Since problems in the data are easier to fix than problems in the code, debugging is easier and faster. Debugging is also helped by the following strategies:

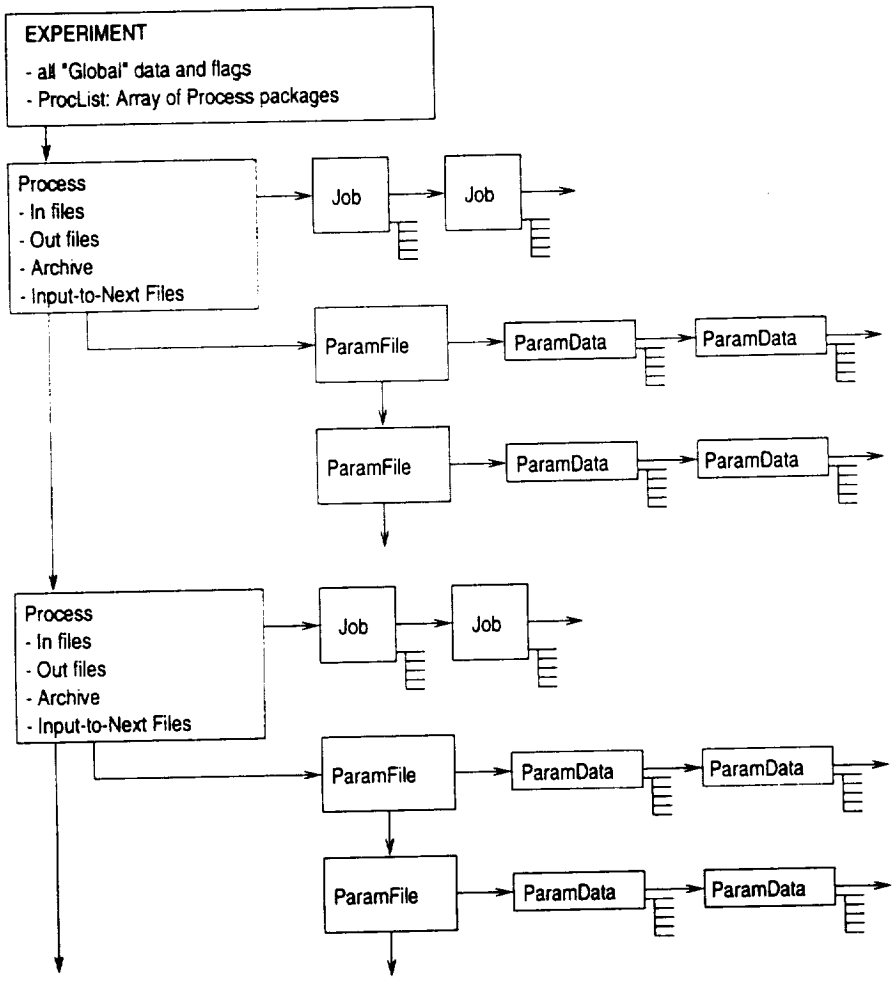

Figure 2: ILab data structures

(1) each package has a "dump" function to print out all variables; (2) each package error-traps the setting of any variable inside the `set` portion of a `get/set` function. A caveat is appropriate here: data duplication is by no means a good or dependable strategy, unless it is closely integrated with code design. This integration means that a programmer must continually review the data members of packages, moving data members as appropriate in order to avoid inconsistencies and incoherences in the code.

As much as possible, we avoided the use of inheritance in order to keep the code structure simple and legible. Some of ILab's dialog packages are derived from existing Tk packages, but this derivation is always merely one-level deep, and is fairly transparent. The only data structure that needed to use inheritance is our `JobModel` package, since the following conditions are true: (1) we have several different "job models" already, and they have enough in common, and enough differences, to justify the existence of a base class; (2) we know that we will need to add more derived job models in the future, as ILab will be expanded to accommodate more meta-computing environments.

Perl is well-known to be a highly flexible language. We were able to further simplify our packages by naming the package members, and the `get/set` functions, by the same names, since the Perl interpreter distinguishes the variable from the function by the calling syntax; the variable is `$reference->{name}`, and the function is `$reference->name`. (NOTE: Perl already makes it easy to collapse `get` and `set` functions into one function, so that instead of two functions accompanying each data member, we only have one.)

Here is an example of this "shotgun" approach in our `Job` package, showing the package `new` (constructor) function, two data members, and the `Status` (the `get/set`) function associated with the `Status` data member:

```
package Job;

sub new {
    my $class = shift;
    my $self  = {};
    $self->{JobID} = undef;
    $self->{Status} = 'NotStarted';
    bless $self, $class;
    return $self;
}

sub Status {
#   Only allow one of six strings for this field
    my $self = shift;
    my $temp = $_[0] if @_;
    if ( defined( $temp ) )     # set the variable if an argument is passed in
        {
        if ( $temp eq 'NotStarted' ||
             $temp eq 'Queued'     ||
             $temp eq 'Running'    ||
             $temp eq 'Stopped'    ||
             $temp eq 'Failed'     ||
             $temp eq 'Done' )
            { $self->{Status} = $temp; }
        else
            { print "attempt to set job status to illegal field = $temp\n"; }
        }
    return $self->{Status};     # always return the variable for get function
}
```

Over the course of a large program, this "same-name" model results in fewer places where bugs can occur and fewer places where the programmer has to reference another part of the code in order to make sure that names are correct.

For those of our packages that need to be made into base packages (for derivation of mostly similar but slightly different packages), we extend the object-oriented approach by putting the package variables into a "closure", thereby making these data members less accessible to programmatic users of the base class.

7 Basic Assumptions and Requirements for Distributed Processing

We have started with several basic assumptions about the distributed computing environment into which jobs will be launched. The first and most important of these is the necessity of maintaining production level capability. This has important implications, the first of which is that all compute intensive application processing must occur under the aegis of a job scheduler and queuing system. This is because modern computer centers require jobs to start through a scheduler, and because any other manner of submitting to shared computational resources would violate our "good neighbor" policy. The second assumption about our distributed computing environment is that it should be able to leverage the Globus metacomputing middleware currently being developed at Argonne National Laboratory. It must also be possible for parameter study users to be able to bypass the Globus layer and still submit jobs into a distributed environment. This has resulted in a parameter study GUI design which incorporates several "job models" for spawning parameter studies in a distributed fashion.

8 Job Models

We will describe the job models in order of their increasing complexity. The simplest of these job models represents an entirely local capability, i.e., all jobs resulting from the creation of the parameter study are also submitted for computation onto the local machine. These job runs occur without the assistance of any scheduler, but may include a parallel job launcher such as "mpirun". ILab generates a single shell script for each run in the parameter study. Each

contains the shell code to construct a main directory for the parameter study, if one doesn't already exist, and then to build its own subdirectory, uniquely named with an automatically generated parameterization identifier. Files required for input by the user's compute executable are copied into the respective subdirectories. The executable is then started. Because no scheduler is assumed, jobs are run sequentially to avoid oversubscribing the local system.

The second job model launches jobs onto a cluster of machines (which may include the originating machine), on each of which the user has an account and an appropriate ".rhosts" file. Each job is implemented with a pair of shell scripts, the first of which remote-copy's ("rcp") the second script to the remote location and then remote-shell's ("rsh") this script on the remote machine. It is the second shell script that does the work of organizing and creating the directory layout on the remote host, and which then starts the chain of computation.

The third job model is similar to the second, except that a scheduler is now assumed. When the scheduler is PBS, the first shell script submits to the scheduler a script containing PBS directives followed by shell commands.

The fourth job model assumes that the Globus metacomputing middleware is used for remote job submission and file manipulation and that a job scheduler (PBS) is used for queuing and starting jobs. The remote script is similar to that of the third job model.

Note that none of the above required shell scripts need to be provided by the user; all shell scripts are automatically generated by ILab during the script generation phase.

In each of the above cases, a parallel job loader (currently MPI is supported) may be specified.

Note that currently files are not cached onto the remote systems at the time of job submission. We have assumed that users will be submitting into a production level environment, and that routing through a job scheduler will be required by the computer center administration. This implies that the most heavily used job models will be the third and the fourth. The typical scenario for usage is that a suite of jobs is submitted through a scheduler, and, as such, the compute resource will be shared with other users. Jobs on the scheduler queue will all compute when they reach the top of the queue, and the total time for the entire parameter study to complete will typically be numbered in days, not hours or minutes. This is based on our experience at NASA Ames Research Center and elsewhere and based on the types of parameter studies users now are contemplating. Furthermore, users will typically be submitting their jobs to run on volatile scratch file systems. This is a requirement since most modern computer centers do not make available to users enough permanent file system space to accommodate the input and output files of substantial parameter studies. Thus, caching of files is risky because there is no guarantee that by the time an individual job from a paramter study is started by the scheduler that input files that have been cached will not already have been purged by a periodic file scrubber. It is unrealistic to expect otherwise. However, we have devised a method for caching files only when jobs finally start. This includes a method for guaranteeing that (1) only one process can be copying an input file to a cache so that there is no file clobbering during the caching procedure (this involves a lock file during the caching process) and (2) that files which were previously cached, but then subsequently deleted by a scrubber will be re-cached on demand by the client job. We will be adding these capabilites to the third and fourth job models.

A few words on the advantages to utilizing shell scripts. We have taken the position that Unix shell languages are the "lingua-franca" of Unix job control language. Our choice is the Korn shell [ref] "ksh", which is a highly expressive language for constructing sequences of commands, and for error-trapping these commands. With the Korn shell language, background processes and "co-processes" (background processes that can communicate with the parent process) may be easily created. Processes may also be easily monitored and subsequently "killed" if necessary. Another advantage of using shell scripts is that these shell scripts may, if desired, be invoked independently of the GUI. There is no requirement that only the ILAB GUI can start the user's processes. Furthermore, those users that wish to may independently modify the shell scripts for their own purposes. The shell scripts, therefore, are "recyclable" if desired. Since the commands in the scripts are interleaved with output statements, these scripts leave a record of their workings, which is used as a log.

The ILab GUI may in part be described as a user interface that collects information on the locations of the user's executable and input files, and then assembles from this collected information the appropriate shell scripts for running this executable. Thus, an additional advantage of using shell scripts is that it is fairly easy to change existing job models or add new job models. This is accomplished simply by modifying the generating code within the ILab GUI. In order to simplify the addition of future job model to ILab, we used the object inheritance capabilities of Perl to create a base `JobModel` package , and several derived `JobModel` packages (`LocalJobModel`, `GlobusJobModel`, etc.). New derived job models, e.g. for the metacomputing environments CONDOR and Legion, can be easily inserted into the existing code framework. Additionally, as a special case of creating and launching a parameter study, it is possible and easy to use ILab to launch single jobs runs (i.e., a singleton "parameter study") into a local or remote compute environment that may require any of Globus, PBS, and/or MPI. Thus, ILab may be used simply as a "Unix

job control language" script generator for launching single jobs and as such, it greatly simplifies the launching process. This is especially true in the case where a job will be run on a remote system and requires the migration of input files and executable.

9 Parameter Study Example - a Case Study

Up until recently, parameter studies of aerospace vehicle flow characteristics typically utilized 2-dimensional computational fluid dynamics (CFD) solvers. This was partly dictated by limitations in the available compute resources, both in terms of CPU time and memory size. In the past several years, because of the increased availability of multiprocessor parallel machines with multi-gigabyte memories, it has become feasible to create parameter studies based on 3-dimensional CFD codes. Nevertheless, the nature of the current generation of aerospace vehicle flow solvers still makes the overhead for such large parameter studies high. As an example, we chose the Overflow 3-dimensional Navier-Stokes flow solver [ref]. This modern CFD code implements the "overset" grid method (overlapping curvilinear grids exchange interpolated boundary information at each time-step). The code is a parallel code and it groups neighboring grids for solution onto individual processors. We applied the Overflow code to the solution of the flow field of the X38 Crew Return Vehicle (CRV), a NASA space vehicle designed as an astronaut "escape pod" potentially for the joint effort space station. Figure 3 depicts the X38 CRV and several of the body-fitted curvilinear grids which define the body surface. The problem we chose was to create a parameter study for two significant flow variables in

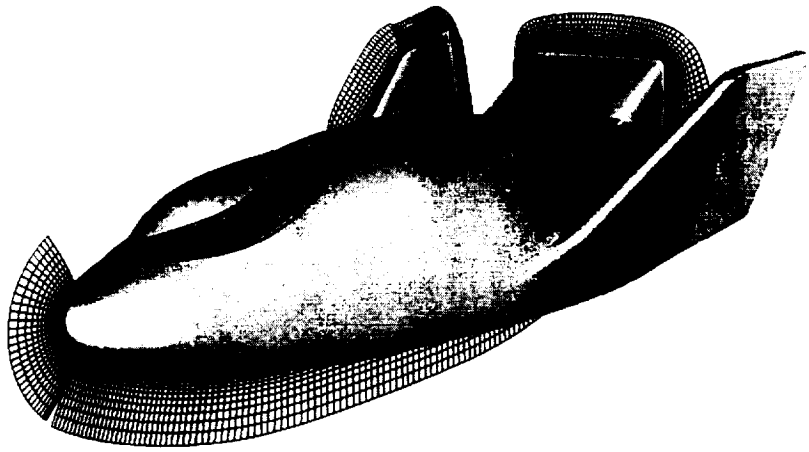

Figure 3: X38 Crew Return Vehicle with several of its computational body grids

a portion of the glide regime of the X38 CRV. In particular, we chose to parameterize in Mach number (basically, the vehicle velocity) and Alpha (the "angle-of-attack"). This results in a two-dimensional parametric study. The geometry of the X38 CRV is defined by 13 curvilinear "body" grids and 115 "off-body" grids (which are strictly Cartesian in topology). The total number of grid points in this grid system is approximately 2.5 million and since the Overflow code requires some 40 double-precision words of memory per grid point, this results in a total memory requirement per run of approximately 800 Megabytes. The Overflow flow solver utilizes the MPI message-passing parallel library. Each run for the X38 vehicle requires four processors.

We used ILab to create a 16×12 parameter study in Mach number and Alpha. Using the ILab tool involves the following steps. First, the user must supply a name and directory for his "experiment", which is where the records for this parameter study will be kept. Then, the names of the local and remote machines on which the runs will occur must be supplied. Next, an input file directory is named, and the input file(s) to be parameterized are specified. At this stage, the parameterizable input file will be displayed in the special-purpose parameterizing editor. The user parameterizes this file as described in the parameterization section above (section 3) and the 192 parameterized input files are created. Next, the user identifies the executable name and location (Overflow, in this case) and also a directory name in which the run subdirectories will be located on each of the executing machines. Options for specifying MPI, Globus, and PBS, and number of processors (4 per run, in this case) are set. At this point, the appropriate shell scripts are generated and are then started. This entire entry process takes under five minutes, and if starting from a previous experiment, a

"most-recently-used" file may be selected. ILab permits the user to make appropriate modifications to the file using the same entry widgets that are used to create a new experiment. For cases like that described above, it usually takes under a minute to create and start an entire parameter study.

In our experiments, two machines were selected for running the jobs, each of which supported MPI, Globus, and PBS. The scripts that were generated by ILab conformed to our fourth job model. ILab then submitted all jobs to the PBS queues on the selected machines and over the next 24 hours (approximately), all jobs completed. From the resulting solutions, we computed forces and moments for the X38 CRV, and from these, we constructed a plot of coefficient of lift over drag for the vehicle. This is displayed in Figure 4. Every point in the lattice represents a complete flow solution.

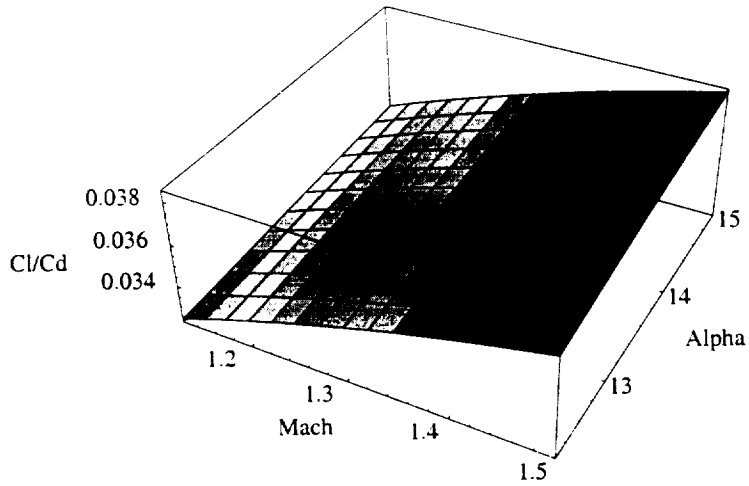

Figure 4: Coefficient of lift over drag for the X38 CRV

10 CAD Tool Process Specification

Currently, all information describing the user's process is collected through a series of "previous-and-next-wizards", which guide the user through the process specification procedure. Though this model is acceptable for single stage parameterizations, it quickly becomes inadequate for specifying complex user processes. These complex processes may include several stages of parameterization, pre- and post-processing of data, archiving of data, resubmission and restarting of user programs, feedback loops to accommodate multidisciplinary optimization, etc. Currently, we are building a visual capability for complex process specification which will be an alternative to the wizard mechanism. This will consist of a CAD tool for constructing a data-flow diagram which describes the user's set of processes. The user will be able to construct this diagram by choosing individual process element icons from a palette and placing them in the diagram by "drag'n'drop" mouse operations. This palette of icons will represent the basic process building blocks such as input file parameterization, moving, copying, or renaming files, running an executable, etc. At each node of the diagram, right mouse button will pop-up an appropriate dialog that will query the user for the details required for that particular operation. What results internally is a directed graph representing the entire set of processes in user's "Experiment": for example, a parameterization followed by the execution of a simulation program, then followed by the execution of post-processing program(s) and archiving. Subsequently, this directed graph is interpreted, and the required individual shell scripts are constructed from the information stored at each node. The construction process consists of assembling the required shell scripts from "cliches", which are small groups of ILab-provided shell commands that perform the operation the user has requested.

Figure 5 depicts an example of a multi-stage parameterization process. In the first stage of parameterization, input to a grid generation program ("Gridder") is parameterized, resulting in three input files. After running the grid generator, three grid systems have been created. These grid systems will be part of the input to a flow solver, as will a flow variables input file. It is this flow input file which is subjected to the second stage of parameterization. In the

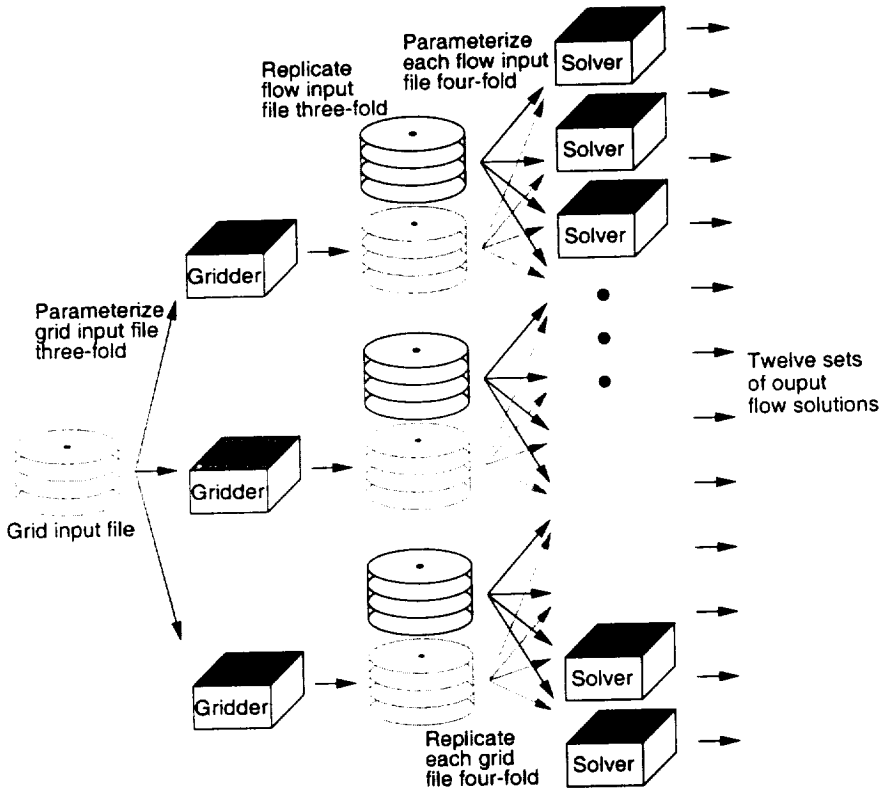

Figure 5: Multi-stage parameterization process

figure, a four-way parameterization has been applied. Each of these four flow input files must be replicated three times to be paired with the three grid files, and each of the grid files must be replicated four times for pairing with the flow input. The result is essentially a two-dimensional parameter study (3 x 4), but it has resulted from two independent stages of parameterization. This adds a higher degree of complexity to the user's process, and consequently, to the mechanisms required for assembling and running these jobs. It is, in part, for this reason that we are constructing a more powerful user interface mechanism for specifying and creating parameterization processes.

Summary

The needs of our user community have triggered the development of ILab, a flexible parameter study creation and job submission tool. This modern GUI implements a modular experimental workbench for programming research into local and remote job submission methods, complex user process specification technology, and for experimentation with Information Power Grid middleware. Our choice of Perl/Tk as a rapid-prototyping development language strongly facilitates these experimentation and anticipated further expansion of the core user GUI capabilities. We have proven our ILab product with significant parameter study computations in a distributed environment. We are currently working closely with users whose parameter study requirements are demanding. We are adding these new capabilities to the tool using an advanced CAD-based user-interface technology.