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

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

Computer Sciences Corporation, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035, USA
{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 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 seeking to combine several successive stages of parameterization and computation. Simultaneously, grid-based computing offers immense resource opportunities but at the expense of great difficulty of use. We present ILab, an advanced graphical user interface approach to this problem. Our novel strategy 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. The compute engines are frequently large parallel machines with multi-gigabyte memories and terabyte disk farms. Researchers have available the resources not only of their own laboratories but also those at other computer centers accessible via fast networks. Parameter studies are now quite feasible and are being performed on a regular basis by researchers who require solution information throughout a given aerospace vehicle flight regime. The difficulties, however, have shifted to the manual creation of these parameter studies and to tasks 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. They produce as output large collections of data files. Currently, most parameter studies are performed with two-dimensional flow solvers, but three-dimensional solvers are also beginning to be used.

Recent developments in grid-based "metacomputing" such as Globus [1]. and Legion [2] 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: the proliferation of *job control language* (JCL) to support these capabilities. This has placed an onus on users of these metacomputing grids, who are typically engineers and researchers not well prepared or enthusiastic about learning or creating the requisite control language scripts for managing distributed parameter studies. NASA is currently building a national metacomputing infrastructure, called the "Information Power Grid" (IPG) [3], intended to provide ubiquitous and uniform access to a wide range of computational, communication, data analysis, and storage resources, many of which are specialized and cannot be replicated at all user sites. However, the interface to the IPG is still under development.

We briefly describe the notion of a "parameter study" by giving two general examples. Simulation codes produce solutions to scientific or engineering problems for some set of input values ("parameters"). Varying these parameters through some prescribed range (the "parameter space") yields a set of related results, called a "parameter study" (sometimes written as "parametric study"). As a second example we point to Monte Carlo simulations. Monte Carlo codes are typically run many times in order to to produce statistically meaningful ensemble averages. This too can be considered a parameter study, where the parameter to be varied is merely the seed for random number generation, and does not actually have any physical significance.

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.

We conducted a literature survey to identify existing parameter study capabilities that fulfilled the need of users at NASA Ames Research Center. The only tools deemed applicable for these tasks were the historically related Clustor and Nimrod codes [5]. Both are able to generate and launch simple parameter studies. They also implement an internal "meta-language" for describing parameter study creation. Additionally, they make it easy to parameterize command line arguments.

However, they did not fully meet the requirements of our users. Some of these are as follows. Users must have access to multiple job submission environments. These must include any combination of PBS [6], LSF [7], MPI, Globus, Condor [8], and Legion. Also, users require the ability to create what we call "multi-stage" parameter studies (a detailed example appears in section 7). 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. Job submission should continue autonomously, and without the continued presence of the parameter study tool. Users also require a fairly comprehensive level of job auditing and scientific diary capability, the secretarial side of a problem solving environment (PSE). On the development side, we needed to design a parameter study tool that could be easily extended using a high-level rapid-prototyping language (such as Perl). This is because we envision using the tool as a testbed for experiments in parameter study creation models, job submission

models, and complex process specification models. We also need to be able to use the tool to generate shell scripts designed for parameter study job submission and for complex process job submission (visual scripting). It is essential that the script generation process be very flexible.

2 Problem Definition

Creating and launching parameter studies without the assistance of automating tools is laborious, tedious, and error-prone. Examining the stages of this task allows us to discern the nature of the inherent problems. The first stage is to create the parameterized input files which incorporate the sets of values representing the parameter study. These sets are the Cartesian product of the individual sets of values over which each of the parameters of interest varies. The total number of combinations (the *parameter space*) can quickly get to be very large, and creating these sets of input files manually is time-consuming and error-prone. Each of the resultant input files represents a run of the user program (a job). Launching jobs involves setting up partitioned file spaces in which they can run, supplying each with all required input files, submitting them, and then monitoring progress and managing output. Our first design requirement was that all of these functions be automated and integrated into a single Graphical User Interface (GUI). The second requirement was that of simplicity of use. We believe that users are very sensitive to ease-of-use issues, and that they will avoid process automation tools that are deemed difficult or non-intuitive. The third requirement is that a parameter study tool be able to self-document its actions. If it cannot, users will quickly be mired in a morass of hundreds, even thousands, of old runs whose origin and purpose are no longer obvious; a complete parameter study tool must be part PSE and part scientific diary. The fourth requirement is that of job submission flexibility in a scientific computation environment currently in flux. This is because "Grid-based" computing has added new complexities and layers of JCL to the task of submitting jobs. ILab meets all of these four user requirements.

3 Basic Assumptions and Requirements for Distributed Processing

We have started with two basic assumptions about NASA's distributed computing environment into which jobs will be launched. The first is the need to maintain production level capability. This has significant implications, because all compute-intensive application processing must occur under the aegis of a job scheduler and queuing system. Any other manner of submitting to shared computational resources would violate "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 bypass the Globus layer and still submit jobs into a distributed environment. This has resulted in a design incorporating several job models for spawning parameter studies in a distributed fashion.

4 ILab: The IPG Virtual Lab

We describe important features of the ILab parameter study tool, in particular parameterization operations and aspects of the internal coding design.

4.1 Parameterization of Program Input Files

In order to minimize the difficulty of building a set of parameterized input files, ILab includes an integrated, special-purpose text editor. This editor has unusual capabilities: it allows the user to mark *graphically* the appropriate parameter data fields and to designate the set of values for each selected field. This parameterizer is depicted in Fig. 1. Value sets can be specified either as a list or by min/max/increment. The user first se-

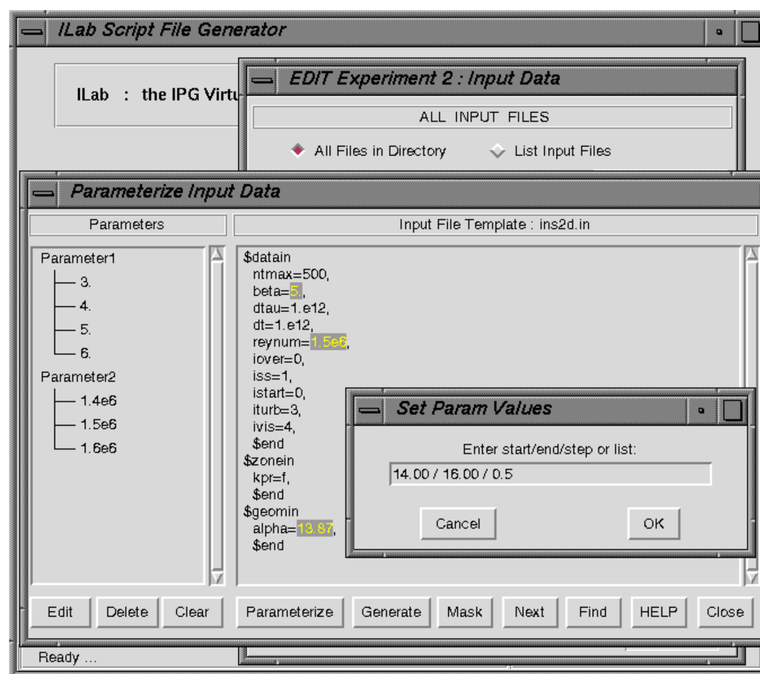

Fig. 1. ILab parameterization screen

lects (highlights) with the mouse those ASCII text fields within the input file which will be parameterized. In Fig. 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 “timestep” for each of several related discretization zone input files), that can be indicated at this stage. After text selection of the appropriate fields, the user enters a

list or range of values for the selected fields. Lastly, the set of parameterized input files is generated. These files constitute the Cartesian product of the individual parameter sets. As an 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 a total of $4 \times 2 \times 3 = 24$ parameterized files will be produced.

Because the file parameterizer is integrated within the ILab GUI and because its use is intuitive, the process of parameterization of the input files has been made trivial. Additionally, a “most-recently-used” (MRU) capability saves the current parameterization state for future reference and for reuse or modification.

4.2 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 require this ability when 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 input files and their associated script files. ILab’s “Edit Parameters” screen - the special purpose editor described in section 4.1 - has a pop-up dialog for this purpose. Users can enter any number of masking rules, and each rule must specify two or more parameter 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 masking rule would be entered as:

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

This syntax, which is the same in Perl, C, C++, and Java, was chosen since 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 Fig. 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 minimal parsing, and use it to delete job objects from the user’s list of experiments.

4.3 Coding Model and Language Choice

We have chosen to build our ILab 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 [9], a free software tool available from Sun Microsystems. 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.

4.4 Object-Oriented Data Structures and Strategies

We used Perl "packages" (the equivalent of classes in C++ and Java) to hold all ILab data, both persistent and transient. Fig. 2 depicts the data structures hierarchy.

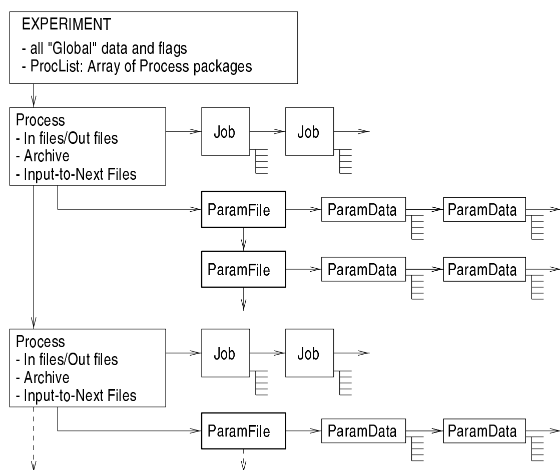

Fig. 2. ILab data structures

An `Experiment` package holds all persistent data: the data is serialized (written *en masse*, retaining data structure hierarchies) to and from disk with the use of Perl's `Data::Dumper` module. To reduce the size of the `Experiment` package, several other arrays of packages apportion data that has to be held in lists or arrays: a `ParamFile` package for each input file to be parameterized, a `ParamData` package for each variable 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. To run a user's parameter study Experiment, a list of `Job` packages is created: some `ParamFile` and `ParamData` data is transferred to Job packages, and additional data is attached. The organization of data in `ParamFile` and `ParamData` is "orthogonal" to the way the same data is organized in the array of `Job` packages: this simplifies script creation, submission, and monitoring. Essentially, data is in arrays of arrays during editing/creation, while during submission/monitoring the same data is flattened out into a one-dimensional 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, which holds 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 a user can make changes and then cancel the changes without 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. Problems in the data are easier to find and fix than problems in the code. Debugging is also facilitated by the following strategies: (1) each package has a "dump" function to print out all variables and (2) each package error-traps the setting of any variable inside the `set` portion of a `get/set` function. Caveat: data duplication is not a good or dependable strategy, unless it is closely integrated with code design. This integration means constantly reviewing the data members of packages, and moving data members as appropriate to avoid inconsistencies and incoherencies in the package design.

To keep the code structure simple and intelligible we avoided as much as possible the use of inheritance. Some of ILab's dialog packages are derived from existing Tk packages, but this derivation is only one level deep, and fairly transparent. The only data structure that requiring inheritance is our `JobModel` package, because (1) we have several "job models" already, and they have enough similarities and differences to justify the existence of a base class and (2) more derived job models will need to be added in the future, as ILab is expanded to accommodate more meta-computing environments. The various job models are described in section 5.

Perl is a highly flexible language. We were able to further simplify our packages by giving the package members and the `get/set` functions the same names, since the Perl interpreter distinguishes the variable from the function syntactically; the variable is `$reference->{name}`, and the function `$reference->name`. Note that Perl already makes it easy to collapse `get` and `set` functions into one function, so that only one function accompanies each data member.

Here is an example of this approach in our `Job` package, showing the `new` (constructor) function, two data members, (`JobID` and `Status`) and the `Status` (`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 variable if argument 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 "illegal job status = $temp\n"; }
        }
    return $self->{Status}; # get func. always returns variable
}
```

In a large program this "same-name" model reduces the number of occurrences where the programmer has to reference another part of the code to ensure that names are correct. For those 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 programming users of the base class.

5 Job Models

We describe the various job models that ILab currently supports in order of increasing complexity. The simplest represents an entirely local capability, i.e., all jobs making up the parameter study are submitted for execution on the local machine. The runs occur without the assistance of a scheduler, but may include a parallel job launcher such as “mpirun”. ILab generates for each run in the parameter study a single shell script, which constructs a main directory for the whole study (if one doesn’t already exist), and then builds its own subdirectory, uniquely named with an automatically generated parameterization identifier. Files required for input by the user’s 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. This is accomplished by chaining the shell scripts: the first script does its work and then submits the next script in the chain, etc. This chaining proceeds even if some command within a script fails (e.g., the user’s primary compute executable).

The second job model launches jobs onto a cluster of machines (which may include the originating machine), on which the user has accounts and an appropriate “.rhosts” file. Each job is implemented with a pair of shell scripts. The first remote-copies (Unix “rcp”) the second script to the remote machine and then executes (Unix “rsh”) it there. It is the second shell script that creates and organizes directory layout on the remote host, and which starts the chain of computation. This job model currently makes no use of schedulers. We have not built in any mechanism for limiting the number of concurrently running jobs on any individual resource. This implies that the individual compute resources may become oversubscribed. We are planning to add a non-scheduler-based job “limiter” into this job model.

The third job model is similar to the second, except that the presence of a scheduler is 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 scheduler (PBS) is used for queuing and starting jobs. The remote script is similar to that of the third job model.

None of the above shell scripts need to be provided by the user; they are automatically generated by ILab. In each of the above cases, a parallel job loader (currently MPI is supported) may be specified.

Currently files are not cached on the remote systems at the time of job submission. We assume a production level environment requiring routing through a job scheduler. This implies that the third and fourth job models will be the most heavily used. The typical usage scenario is that a suite of jobs is submitted through a scheduler, and that the compute resource is shared with other users. The time for the parameter study to complete will often be numbered in days, not hours or minutes. This is based on experience at NASA research centers and on knowledge of the types of parameter studies users are contemplating. Such computations frequently utilize volatile scratch file systems when user allocations of permanent file-system space are insufficient to accommodate the input and output files of substantial parameter studies. Advance copying of files is therefore risky, since cached input may have been purged by a file scrubber by the time an individual job is started by the scheduler. However, we have devised a method for

just-in-time caching. It guarantees that (1) only one process copies an input file to a cache, which avoids clobbering (involves lock file), and (2) that files previously cached, but subsequently deleted by a scrubber, are re-cached on demand by the client job. We will add these capabilities to the third and fourth job models.

Utilizing shell scripts has several advantages. Unix shell languages are the "lingua-franca" of Unix JCL. Our choice is the Korn shell [10], a highly expressive language for constructing sequences of commands, and for error-trapping them. In the Korn shell, background processes and "co-processes" (background processes that can communicate with the parent process) are easily created. Processes may also be easily monitored, and killed if necessary. Another advantage of using shell scripts is that they may be invoked independently of the GUI. There is no requirement that only the ILab GUI start user processes. Users may modify the shell scripts for their own purposes; they are"recyclable." The commands in the scripts are interleaved with output statements, which leave a record of their workings which acts as a log.

ILab may, in part, be described as a GUI that collects information on the locations of the user's executable and input files, and assembles shell scripts for running this executable. It is fairly easy to change existing job models or add new ones, which is accomplished simply by modifying the script generating code within ILab. In order to simplify the addition of future job models to ILab, we used the object inheritance capabilities of Perl to create a base `JobModel` package and several derived packages (`LocalJobModel`, `GlobusJobModel`, etc.). New derived job models, e.g. for the metacomputing environments Condor and Legion, can be easily inserted into the existing framework.

It is possible, and easy, to use ILab to launch single jobs (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 convenient Unix JCL script generator for launching single jobs. This is especially beneficial when a job will be run on a remote system and requires the migration of input files and executable.

6 Parameter Study Example - a Case Study

Until recently, parameter studies of aerospace vehicle flow characteristics utilized mostly two-dimensional computational fluid dynamics (CFD) solvers. This was partly dictated by limitations in the available compute resources (CPU time and memory size). Recently, however, because of the increased availability of multi-processor machines with large memories, parameter studies based on three-dimensional CFD codes have become feasible. Nevertheless, the overhead for such large studies remains high. As an example, we chose the Overflow three-dimensional Navier-Stokes flow solver [11], which employs the overset grid method (overlapping curvilinear grids exchange interpolated boundary information at each time-step). The MPI parallel version of Overflow groups neighboring grids for solution onto individual processors. We used Overflow to compute the flow field of the X38 Crew Return Vehicle (CRV), a NASA space vehicle. Fig. 3 depicts the X38 CRV and several of the body-fitted curvilinear grids defining its surface. The complete configuration consists of 13 curvilinear body-fitted grids and 115 rectilinear off-body grids, totaling approximately 2.5 million points. Overflow uses

some 40 double-precision words of memory per grid point, which results in a total memory requirement of approximately 800 MB per run.

We chose to create a 16 × 12 parameter study for two significant flow variables in a portion of the glide regime of the X38: Mach number (normalized vehicle velocity) and Alpha ("angle-of-attack"). This results in a two-dimensional parametric study consisting of 192 runs. Each run for the X38 vehicle requires four processors.

Using ILab involves the following steps. First, the user supplies a name and directory for the "experiment", which is where the records for this study will be kept. Then, the local and remote machines on which the runs will occur are selected. Next, an input file directory, and the input file(s) to be parameterized, are specified. Input files are displayed in the special-purpose graphical editor, and parameterized as described in section 4.1, producing 192 parameterized input files. Next, the user identifies the executable name and location and also a directory where the run subdirectories will be rooted on each of the executing hosts. Options for specifying MPI, Globus, and PBS, and number of processors (four per run, in this case) are set. At this point, the appropriate shell scripts are generated and then initiated. This entire entry process takes under five minutes. If starting from a previous experiment, an MRU file may be selected, permitting the user to make appropriate modifications through the same widgets 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.

Two machines were selected for running the jobs, each supporting MPI, Globus, and PBS. The scripts generated by ILab conformed to our fourth job model. ILab submitted all jobs to PBS queues on the selected machines, and within approximately 24 hours all jobs completed. From the resulting solutions we constructed a plot of the coefficient of lift over drag (Cl/Cd) for the X38 CRV. See Fig. 4. Every point in the lattice represents a complete flow solution.

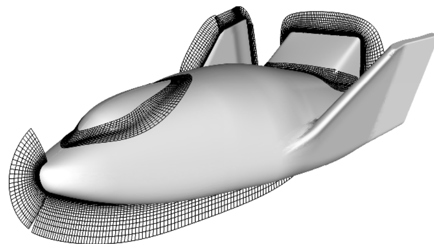

Fig. 3. X38 Crew Return Vehicle with several of its computational body grids

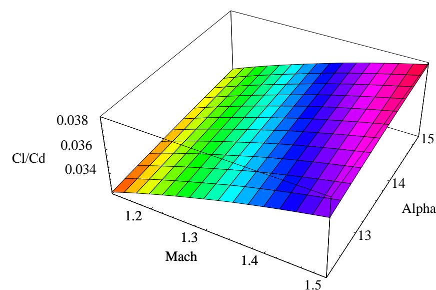

Fig. 4. Coefficient of lift over drag for the X38 CRV

7 CAD Tool Process Specification

Currently, all information describing the user's process is collected through a series of previous-and-next-wizards, guiding 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 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, providing an alternative to the wizard mechanism. It consists of a CAD tool for constructing a data-flow diagram describing the user's set of processes. The user creates a diagram by choosing individual process element icons from a palette and placing them in the diagram by mouse operations. Each icon represents a basic process building block, such as input file parameterization, moving, copying, or renaming files, running an executable, etc. At each node of the diagram a context-sensitive pop-up dialog queries the user for the necessary details. Internally, a directed graph representing the entire set of processes in the Experiment is created, e.g., a parameterization, followed by the execution of a simulation program, followed by the execution of post-processing program(s) and archiving. This 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 macros, small groups of ILab-provided shell commands that perform the requested operation.

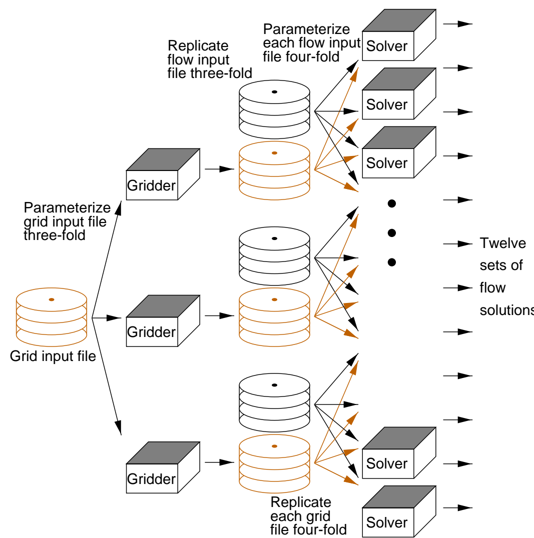

Fig. 5. Multi-stage parameterization process

Fig. 5 depicts an example of a multi-stage parameterization process. In the first stage, 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 (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, 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 × 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 IPG middleware. Our choice of Perl/Tk as a rapid-prototyping development language strongly facilitates 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 ILab using an advanced CAD-based user-interface technology.

References

1. Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. Intl J. Supercomputer Applications, 11(2):115-128, 1997
2. Grimshaw, A., Ferrari, A.,Knabe, F., Humphrey, M.: Legion: An Operating System for Wide-Area Computing. Dept. of Comp. Sci., U. of Virginia, Charlottesville, Va. Available at ftp://ftp.cs.virginia.edu/pub/techreports/CS-99-12.ps.Z
3. NASA Information Power Grid: http://www.nas.nasa.gov/Pubs/NASnews/97/09/ipg_fig1.html and
   http://www.nas.nasa.gov/Groups/Tools/IPG/
4. Clustor (now called Enfuzion): http://www.turbolinux.com/products/enf/enfuzion.html
5. Abramson, D., Sosic, R.,Giddy, J., Hall, B.: Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations. The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995
6. Henderson, R., and D. Tweten, D.: NASA Ames Portable Batch System: External Reference Specification. NASA Ames Research Center, December 1996
7. LSF: http://www.platform.com/
8. Litzkow, M., Livny, M.: Experience With The Condor Distributed Batch System. IEEE Workshop on Experimental Distributed Systems, Oct. 1990, Huntsville, Al. http://www.cs.wisc.edu/condor/publications.html
9. SpecTcl: http://dev.scriptics.com/software/spectcl
10. Bolsky, M. I., Korn, D. G.: The KornShell Command and Programming Language. Prentice Hall, 1989
11. Wissink, A. W., Meakin, R. L.: On Parallel Implementations of Dynamic Overset Grid Methods. Proceedings of SC97, High Performance Computing and Networking, San Jose, CA, Nov. 15-21, 1997