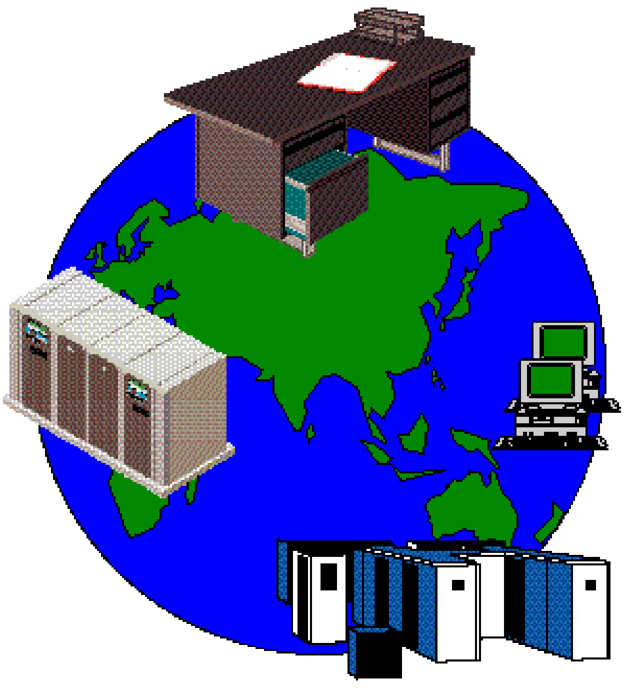
ACTIVE TOOLS™

Clustor: User's Manual

Active Tools P/L

8/11/96



ISBN 0-000-000000-0

CHAPTER 1 Introduction

Clustor is a new generation of tool which brings distributed high performance computing to your desktop. It is designed to provide a very high level interface which allows users to perform scenario based experimentation with little or no programming. The underlying computational platform can contain a mix of supercomputers, high end workstations, or even PCs running a range of operating systems. Clustor provides a graphical user interface which provides a meaningful view of each experiment in terms of the problem specific variables, and it hides the underlying computational platform.

Clustor has been designed to improve the productivity of computational scientists and engineers. Input parameters are controlled through specially tailored graphical user interfaces. A wide range of post processing tools can be used so that users can display their results in a variety of formats.

Clustor can be utilised with a wide range of existing applications. In its simplest form, Clustor makes it possible to augment a command like driven program with an interactive graphical user interface. The full sophistication of Clustor allows a user to make use of a distributed set of computers in a seamless manner, delivering very high performance to the desktop.

This manual provides an overview of the Clustor system together with detailed reference material. It gives a number of sample Clustor applications which highlight the individual features of each application.

Scenario based experimentation

Scenario based experimentation occurs when a user runs a *single* application under a wide range of input conditions and then aggregates the results of these different runs for interpretation. This mode of exploration is becoming more common. Here are a few examples of how it can be applied:

- An environmental engineer may wish to perform an impact study on varying the amount of waste sent into a dump site;
- An architect may wish to study the effect of varying the rigidity of a particular beam in a building design and produce a solution which optimises both cost and safety;
- An aerospace engineer may wish to alter the shape of an aerofoil and observe the effect on drag, lift and cost;
- An electronics designer may wish to vary component tolerances and observe the effect on the performance of some analogue circuit;

- An aerial designer may wish to alter the shape of an antenna and observe its gain.

Clustor is the first tool which provides direct support for scenario based experimentation, without requiring the user to write new programs, or construct complex shell scripts. Equally importantly, Clustor provides the high level of performance required for scenario based experimentation through the use of a distributed metacomputer.

Metacomputing

A metacomputer is a distributed collection of computers, potentially located at physically distant sites, that can be assembled to form a logical parallel computer. This logical computer can be used to access unique resources not accessible at a particular site, or to assemble aggregate computational resources superior to that offered by a single site. In principle, metacomputing can both increase accessibility to supercomputing capabilities and provide more cost-effective computing than conventional high-performance systems.

Currently, there are very few high level tools for programming metacomputers. In order to create an application for a metacomputer today, a user has to build a parallel program, using a range of parallel programming toolkits and environments. This process requires advances programming skills and a good deal of determination. Clustor solves this problem by targeting scenario based experimentation as an application focus.

Why do I need to use a metacomputer?

Scenario based experimentation can require enormous amounts of processor time. It is not uncommon for such models to require in the order of hours to evaluate any one set of parameters. If only three variables are explored, with 4 values per variable, 64 different simulations must be performed, and this can amount to days of workstation time. Metacomputers have the potential to provide the necessary computational resource. Their components, often conventional workstations, are inexpensive, and are often idle for long periods. However, managing the multiple jobs manually can be complex and time consuming, and thus tools are required in order to harness their power effectively, and to provide a seamless computational platform.

Clustor is versatile. It can work with small configurations, down to a single workstation, up to large networks of machines distributed on wide area networks. It currently works with both the UNIX® and WindowsNT® operating systems.

What does a user need to do?

Clustor has been designed so that it is possible to set up computational experiments with little or no programming. Each new experiment is defined by a *plan*, which is a file containing a list of the input *parameters* and a set of *tasks* forming perform the experiment. Clustor manages the construction of a number of tailored graphical user interfaces which are generated automatically, and controls the execution of the programs for each set of parameter values. *Plans* are normal text files, which can be built using a text editor. Clustor also provides a tool for generating plans for common scenarios using a basic *skeleton*. Thus, it is possible to set up a new experiment extremely rapidly, allowing the scientist of engineer to concentrate on designing experiments which are meaningful.

Parameters

Parameters are the key to the way Clustor generates work. The application is executed multiple times, each with a different set of parameter values. Clustor automatically substitutes formal parameter placeholders for actual values at run time. The placeholders can be specified in a variety of positions, including the command line of the executable programs and within input files. Clustor begins by producing a list of values for each application variable. It then generates the *cross product* of all of these values, and runs the application for each unique set of values.

What does my computational platform need to look like?

Clustor runs on anything from PCs to supercomputers. It uses a distributed architecture which provides transparent access to a wide range of platforms. In the simplest configuration, the user runs a Clustor process on the same machine as a remote execution process. Clustor automatically connects to the two processes and executes the jobs which are generated sequentially until the entire experiment has been completed. If multiple remote execution processes are started on many platforms, then Clustor executes as many jobs as possible concurrently. In this way, otherwise idle computers can be used to reduce the elapsed time for an experiment. It is possible to configure Clustor to make use of machines available on a local area network, as well as computers distributed across the Internet. Each remote execution process consults local machine resource constraints, so it can be configured to make use of either prime or low grade cycles as required. Workstation owners can further restrict the hours that jobs can be executed.

The Clustor Environment

Clustor supports three main phases of an experiment, the *preparator* phase, the *generator* phase and the *dispatcher* phase.

The *preparator* is an interactive tool which helps a user builds plans for commonly used modes of operation. Specicially, it is targetted at generating *plans* for experiments in which a number of input files are processed by a model, and a number of output files are generated as a result. Plans can also be built by writing a plain text plan file. The *generator* phase takes as existing *plan* and interactively builds a specific experiment. The *dispatcher* phase is responsible for finding computational resources and performing the work.

Plans

Plans are user defined descriptions of how to perform the computational experiment. Plans contain a description of the parameters as well as a set of tasks to be performed. Clustor provides range of commands for executing and controlling programs, and for sending files between the remote execution processes and the controlling machine. Different *tasks* are invoked at each stage of a job’s execution. For example, it is possible to execute one set of commands when a job is first dispatched to a remote machine and when the job is completed.

Files

Clustor does not assume a shared file system across the metacomputer. Instead, files are transferred between the controlling machine and the execution servers as required. Special commands are available for creating, deleting and appending information to files. These commands operate either on the client or the server machine in a consistent manner. It is possible to use the different scripts to *stage* the dispatch of files to remote nodes, so files which are required by a number of jobs can be sent to a remote system once, whereas the files required by individual jobs can be sent just before the program is executed.

Security

Clustor implements a security mechanism which utilises the normal underlying protection mechanism provided by passwords. Servers are started transparently and automatically by a remote login (using the industry standard Telnet utility) to each of the machines named in a configuration file, and thus they inherit the access domain of that user. Passwords can either be stored in a file, or can be entered interactively by the user for each server machine as required. By using an industry standard remote login utility, it is possible to run Clustor on many systems which already incorporate security firewalls, because these firewalls have already been configured to allow such traffic to pass. Consequently, no special privilege is required to install and operate Clustor.

Fault Tolerance

Clustor tolerates failures in a variety of components. If the controlling process fails, then it can be restarted using a special restart command. When this happens, it re-connects to the jobs which are currently executing, and continues without loss of data. If the network between the controlling and any of the remote machines fails, then the controlling machine re-connects once the service is restored. If a remote execution process itself fails, then the jobs is restarted automatically.

The rest of this manual.....

Chapter 2 provides a number of typical uses for Clustor in a tutorial style, and introduces the general syntax and operational semantics by example. Chapter 3 introduces Clustor and its various commands. Chapter 4 shows how to write plans, and Chapter 5 discusses machine configuration issues. Chapter 6 provides system management information.

CHAPTER 2 Clustor by example: A tutorial

In this chapter we will introduce the utility of Clustor through a few examples.

A simple application: A shop simulation

Consider the following example. A user has written a discrete event simulation which models a shop. The model accepts parameters for

- The number of queues
- The number of servers
- The arrival rate of customers
- The average service time

The program (called `shop`) takes these parameters from the command line (using standard Unix conventions), and performs a simulation. It outputs the average number of customers serviced per hour as a simple number on standard output as well as echoing the input parameters. For example, the following command returns the average number of customers serviced per hour (of 8.95) for 5 queues, 6 servers, with customers arriving at 10 per hour and an average service time of .25 hours each.

```
% shop -q 5 -s 6 -a 10 -s 0.25
8.95 5 6 10 0.25
%
```

Using this model we wish to explore the behaviour of the system as the key parameters are varied, in particular, we wish to explore:

- up to 6 different numbers of queues
- up to 7 different numbers of servers
- up to 10 different arrival rates
- up to 5 different service times.

The user wishes to explore which parameter setting maximises the number of customers serviced. This simple exercise generates 2100 jobs, which at one minute per job would take 35 hours on a single computer. However, if 35 workstations are available, Clustor can reduce the elapsed time to about one hour

How would we solve it manually?

One way of solving this problem is to set up a command file with the following commands:

```
shop -q 1 -s 1 -a 10 -s 0.1
shop -q 1 -s 1 -a 10 -s 0.1
      •    •    •    •    ••
shop -q 9 -s 1 -a 10 -s 0.1
```

The output of this command is then sorted and the minimum extracted. It is obvious that is approach is very time consuming and error prone, and does not lend itself to easy modification of the values. Also, it does not exploit distributed computers which are idle without a significant amount of modification. Of course, it is possible to create scripts which contain loops, or to write pre-processor programs which generate the required command files. Clustor performs all of these tasks in a way which is easy to comprehend and modify. As discussed before, Clustor is controlled by simple user defined *plan*.

Setting up a plan for the shop problem

It is extremely easy to set up a plan for the shop example using the Clustor preparator. The shop experiment makes use of no input files, and thus it is only necessary to instruct Clustor about the command required to run the shop program, and also the format and type of parameters involved.

To start Clustor, the following command is required:

```
clustor
```

Following this, a window is produced giving the user the option of running the preparator tool, the generator tool or the dispatcher tool. Select the preparator. Select the tasks menu item, and fill in the main task as shown in Figure 1. As shown, standard output is redirected to a file called output. This file is then copied back to the root machine, with a job identifier added to the file name. After the run is finished, all of the output files are concatenated to form one file called output. Press apply, and the plan statement will be shown in the main preparator screen window.

Now, select the parameters menu item, and create new parameters *numq*, *nums*, *arrivalr* and *servtim*. A sample for the parameter numq is shown in Figure 2. After four parameters have been created, the resulting plan is created, as shown in Figure 3. It contains commands which inform Clustor of the names of the parameters, plus the tasks which must be run at various stages of the experiment.

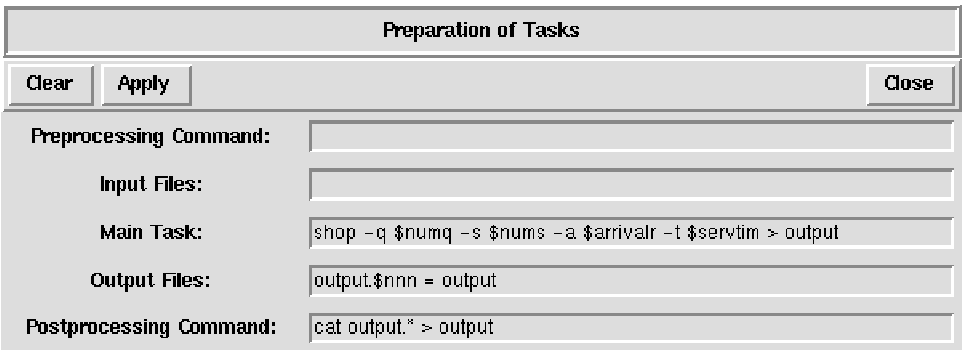


FIGURE 1. **Preparator Task Screen for Shop example**

In this example, 4 parameters are created, called *numq*, *nums*, *arrivalr* and *servtim*. These names can be instantiated in the body of any plan. Thus, in the job task, they are placed on the command line, preceeded by a $ sign, of the shop program, and will be replaced at run time by the correct values.

The main task informs Clustor to run the shop program and to send the output to a file called output. After running the program, output is copied back to the controlling machine, renaming each instance of the output file with a unique extension.

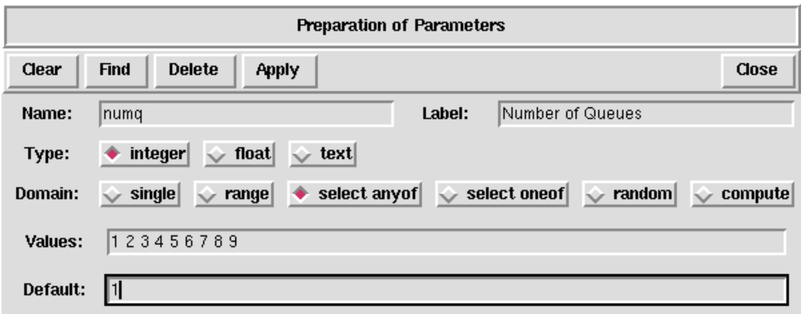


FIGURE 2. **Preprator Parameter Screen for Numq parameter**

When the job is complete, all of the output files are concatenated using the Unix cat command into one file.

parameter numq **label** "Number of queues" **integer**
select anyof 1 2 3 4 5 6 7 8 9 **default** 1;
parameter nums **label** "Number of servers" **integer**
select anyof 1 2 3 4 5 6 7 8 9 **default** 1;
parameterarrivalr **label** "Arrival Rate (/hour)" **integer**
range from 0 **to** 100 **points** 1;

```
parameter servtim label "Service Time" float
          range from 0 to 5 points 5;
task main
  remote:execute shop -q $numq -s $nums -a $arrivalr
          -s $servtim >output
  copy remote:output output.$nnn
endtask
task runfinish
  execute cat output.* > output
endtask
```

FIGURE 3. **Script for the Shop Problem**

Setting up a machine configuration file

Before Clustor can be run a special file (called clustor.nodes) must be created which contains the identity of the machines which are to be used. this file contains the Internet name of the machine as well as the user name and password to be used for each machine. If the passwords are omitted, then Clustor prompts for a password through standard input. the following sample file will make use of two different machines. The following text shows the contents of a configuration file which will execute servers on 2 computers called indy1.cit.gu.edu.au and indy2.cit.gu.edu.au. The file does not contain passwords for user *davida* on the two machines, which must be entered when Clustor is executed.

clustor.nodes

```
indy1.cit.gu.edu.au     davida
indy2.cit.gu.edu.au     davida
```

Generation

Once the plan created by the preparator has been saved to a file, the generation phase can be started by pressing the generator menu item under the Clustor main menu. The screen shown in Figure 4 is produced. The user enters the parameter ranges discussed, and Clustor indicates that 2100 jobs will be created as a result.

Under the FILE menu item, the Generate command is selected, which produces a prompt for a job name as shown in Figure 5. This name is used to identify the particular experiment being performed.

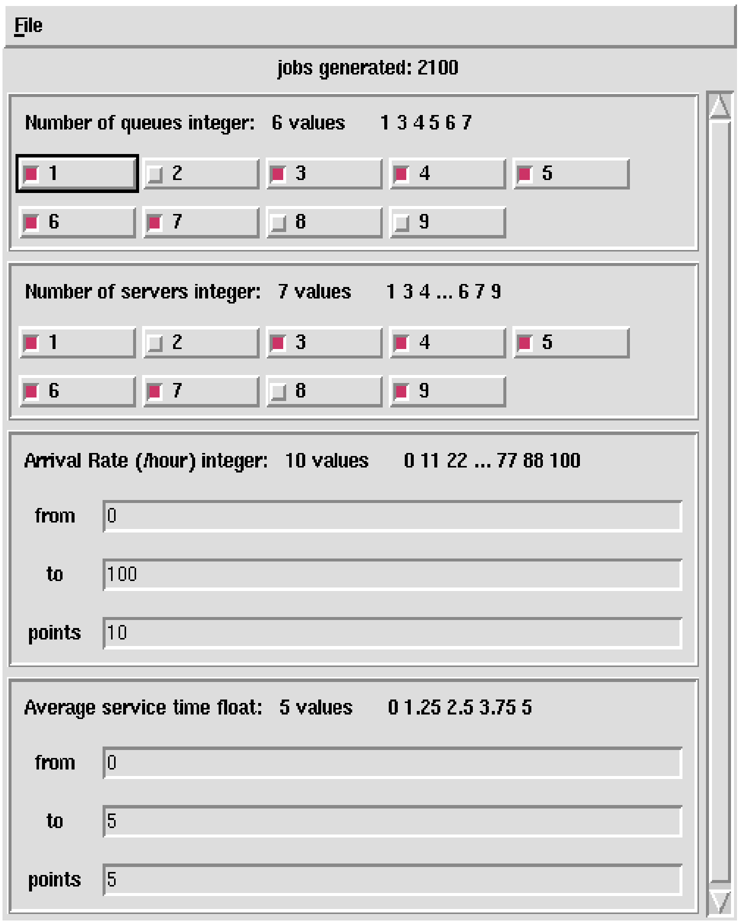


FIGURE 4. **Generation screen for the shop example**

FIGURE 5. **Prompt for run file name**

After a file name has been entered, the dispatcher phase is invoked using the Clustor main memu item.

Dispatch

The dispatch phase is responsible for sending jobs to remote execution servers and displaying the state of the experiment.

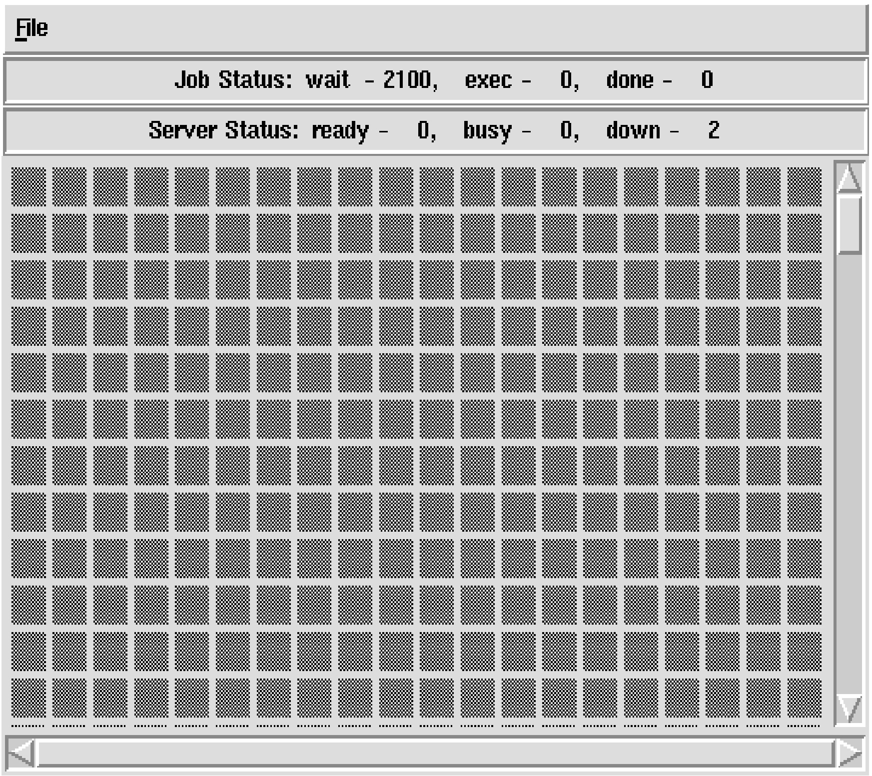


FIGURE 6. Dispatcher screen for shop example

Figure 6 shows a sample screen for the shop example. In order to start the experiment, the File menu item START is selected. Clustor will automatically show the progress of the experiment, and will stop when all jobs have been completed. At the end of the run, a file called output (see the script) is created containing the standard output of all of the shop runs.

Finding the maximum throughput?

The maximum can then be easily selected by sorting the output file and taking the one which maximises the number of customers services. For example, under Unix, the following command is effective:

% sort -n output | tail -1

Something more complex

The example given given to date has the following characteristics:

- The program reads its parameters the command line
- No input files are required
- The program output is returned in standard output

We now explore a more complex example in which:

- A number of shared input files are required by all instantiations of the model
- Each instantiation of the model requires an input file containing the various parameter values
- A number of different output files need to be returned

An Advanced Simulation

Imagine that the example given in the last chapter is changed so that the shop simulation program reads a number of input files before it performs the simulation. Imagine also, that the parameters which control its execution are no longer read from the command line, but are taken from a structured input files. Finally, imagine that the output is written to a file called “outputfile”.

This simulation nows requires Clustor to send some files to the remote node when it first starts, and a parameter input files must be built before the job can be started. Finally, when the run is complete the files must be transfered back to the root machine.

The plan shown in Figure 2 contains commands to control this more advanced simulation. In this example, the parameter statements are the same as those in Figure 2. An additional task is added to execute at node start time. This task copies a number of input files to the remote file system, which can then later be accessed by the program “shop”. The job task is modified to take account of processing the input files containing the parameters.

```
parameter numq label "Number of queues" integer
        select anyof 1 2 3 4 5 6 7 8 9 default 1;
parameter nums label "Number of servers" integer
        select anyof 1 2 3 4 5 6 7 8 9 default 1;
parameterarrivalr label "Arrival Rate (/hour)" integer
        range from 0 to 100 points 1;
parameter servtim label "Service Time" float
        range from 0 to 5 points 5;
task job
  copy skeleton remote:skeleton
```

remote:**substitute** remote:skeleton remote:input
remote:**execute** shop
copy remote:output/* output.$nnn/*
endtask
task nodestart
copy input/* node:
endtask
task runfinish
execute cat output.* > output
endscript

FIGURE 2. Script for the Shop Problem

The contents of the skeleton file are show in Figure 3. Here the file contains place holders for each of the parameters, one per line with the correct keywords for the application also on the line. The substitute command produces a new files, called "input" which has the actual values for the parameters instead of the place holders. The task first copies the skeleton file to the node and then performs the substitute. This guarantees that each job receives a different copy of the input file since they are separated in the node file space.

```
NUMBERQUEUES = $numq
NUMBERSERVERS = $nums
ARRIVALRATE = $arrivalr
SERVERTIME = $sertim
END
```

FIGURE 3. Sample skeleton file

CHAPTER 3 Using Clustor

Figure 4 shows the basic flow of a scenario based computational experiment controlled by Clustor. There are three phases, namely *preparation* of a *plan*, *generation* of a *run* and *dispatch* of the *run*.

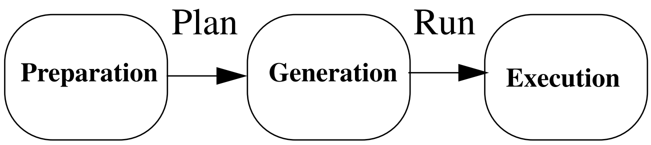


FIGURE 4. Flow of a Computational Experiment

Preparation

Preparation involves creating a plan file which contains the basic information about the experiment, namely the parameter specifications and the tasks which must be performed. Because plans are written in a textual language, they can be created either by a text editor, or by a special GUI based preparation tool. Clustor supports both approaches. A user can type in a plan file and submit it to Clustor directly, or can use a tool called the preparator to build a plan for commonly executed scenarios.

Generation

Generation involves building a run description file from a plan. Since the plan does not contain specific information about which values the parameters should take, this is added during the generation phase using user input. A graphical user interface is built by Clustor, and the user is able to input the actual values for the parameters. The run file contains the exact commands which must be executed in order to perform the job, including information about how to transfer files between the various computers.

Dispatch

The dispatch phase takes a valid run file and performs the execution and distribution of the experiment. During dispatch Clustor displays the current progress of the experiment, including the number of running, completed and pending jobs.

The Execution Model

Clustor makes use of a number distributed processes to control an experiment. When the user invokes Clustor on a workstation, the machine becomes known as the *root* machine because it controls the experiment. When the dispatcher executes code on remote platforms, each of these is known as a computational *node*. Thus, as shown in Figure 5, a given experiment can be conducted with one *root* and multiple *nodes*.

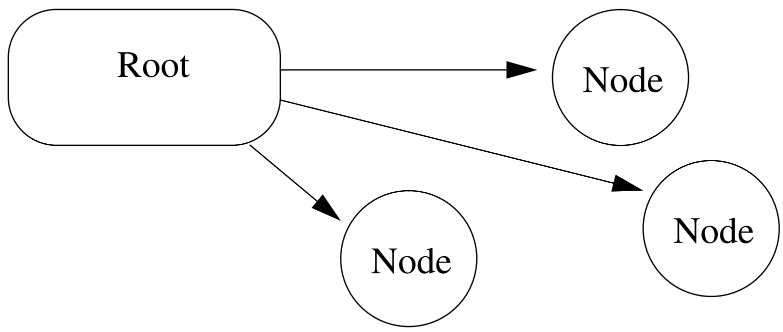


FIGURE 5. Execution Model

Tasks

During execution Clustor runs a number of tasks, which are user defined executable command scripts. Each task contains a number of Clustor commands (as defined in Chapter 4), which in turn can contain executable Unix commands. There are 5 main phases during execution: Root Startup, Node Startup, Job execution, Node Completion and Root Completion. When each phase occurs, the associated task script is run, this a user can perform various operations at specified times. For example a user might wish to create some files when the Clustor root is started, copy those files to each node as it starts, and then run a specific application for each job. Further, the user might wish to copy certain output files back to the root machine when the node completes, and perform some final post processing when the root completes.

Starting Clustor

Clustor can be executed using the following command:

```
clustor file.pln
```

where file.pln contains a valid plan. The screen shown in Figure 6 is displayed, giving the user the choice of invoking the preparator, the generator or the dispatcher.

FIGURE 6. Clustor Startup Screen

The Preparator

The preparator allows the user to build a plan without explicitly writing any Clustor commands. It consists of two key interactive tools:

- one for building parameter statements
- one for building task statements

These tools require a user to fill in information about input and output files, how to run the application, and any pre and post processing commands. The user is then able to save the plan which is generated and this can be edited by a normal text editor.

The Task Builder Tool

Figure 7 shows the main screen for the preparator task building tool. The pre processing field allows the user to enter any executable statement which it to be performed on the root machine prior to starting the experiment. A common use for this is to set up files for a run and perform general preprocessing. The input files field consists of a list of space seperated input file names which must be copied to the remote nodes before the computation can be performed. The Main Task field is used to enter the executable command. The post processing field is used to enter any executable statements which are to be run after the job has completed.

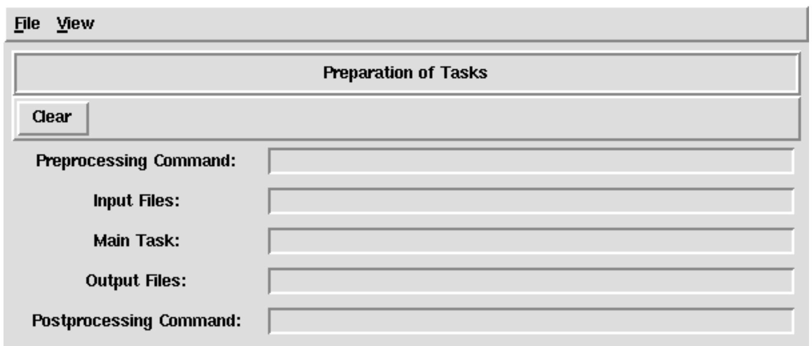


FIGURE 7. Task Builder Tool in Preparator

Figure 8 shows a sample input for the task builder screen and Figure 9 shows the task statements which are produced. In this example a program call

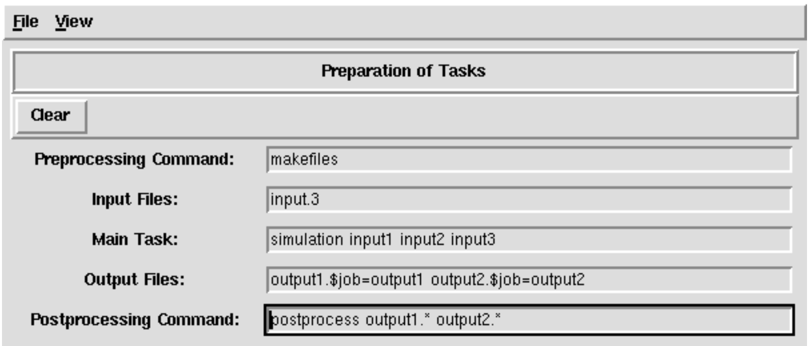


FIGURE 8. Sample Task builder screen

"makefiles" is run on the root machine before the experiment starts, and post processing of the output files occurs on the root machine after they have been returned from the computational nodes. The executable commnd runs a program called "simulation" which takes the three input files, "input1", "input2" and "input3". These three files are copied to the remote nodes before the run can be performed and the output files are copied back after the run. Before they are copied back they are renamed to include a job identifier which makes them unique on the root machine.

task rootstart
    execute makefiles
endtask
task nodestart
    copy input.1 remote:input1
    copy input.2 remote:input2
    copy input.3 remote:input3
endtask
task main
    remote:**execute** simulation input1 iput2 input3
    copy remote:output1 output1.$nnn
    copy remote:output2 output2.$nnn
endtask
task rootfinish
    execute postprocess output1.* output2.*
endtask

FIGURE 9. Sample output plan from preparator

If any of the input files contain parameters, then these can be substituted before the executable command is run. The use of a substitution operation can be performed by a special syntax in the input files field, namely:

<newname>=<skeletonfilename>

In this case, a new file is built which is exactly the same as the skeleton file, except that the formal parameters are replaced by actual values for the run. Since the skeleton file is required on the remote node, it must also be listed as an input file. Given the task generation screen shown in figure Figure 10 the plan in Figure 9 is modified to the one shown in Figure 11.

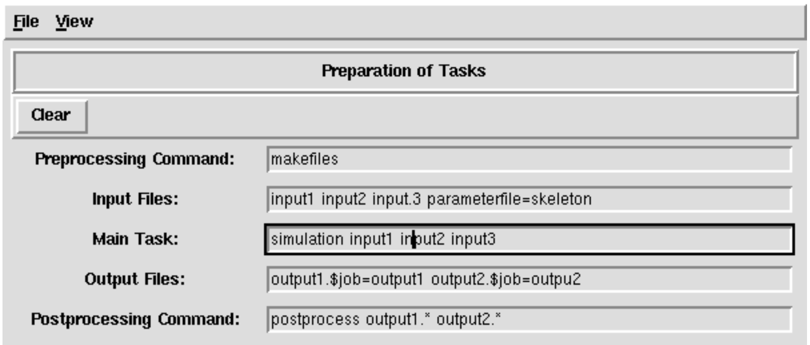


FIGURE 10. Use of file substitution in task builder

task nodestart
    copy input.1 remote:input1
    copy input.2 remote:input2
    copy input.3 remote:input3
    copy skeleton remote:skeleton
endtask
task main
    remote:substitute remote:skeleton remote:parameterfile
    remote:execute simulation input1 iput2 input3
    remote:rename remote:output1 remote:output1.$nnn
    remote:rename remote:output2 remote:output2.$nnn
endtask

FIGURE 11. Sample plan showing use of name substitution

The Parameter Builder Tool

The parameter builder tool allows the user to create parameter statements via a graphical user interface. The tool walks through the various parameter types, and allows the user to select the parameter type, text strings for the generation screen and default values. A sample screen is shown in Figure 12, in which a parameter called

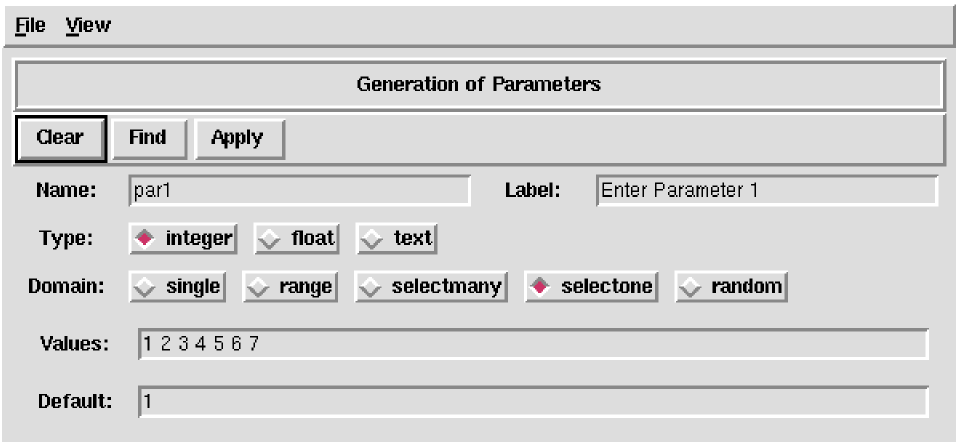


FIGURE 12. Parameter Builder screen in preparator

Figure 13 shows the parameter statement which is generated as a result of this input.

```
parameter par1 label "Enter Parameter 1" integer
        select oneof 1 2 3 4 5 6 7 default 1;
```

FIGURE 13. Sample parameter statement build by preparator

The Generator

If Clustor is invoked with a file name, then then generator immediately displays a screen for controlling the experiment. Each parameter is set using a tailored graphical user interface. After the parameter values have been entered, the generator displays the number of jobs that will be generated. The user can save the resulting run for later use by the dispatcher.

The Dispatcher

This dispatcher accepts a valid run file and allows the user to start an experiment. When the START option is selected, the dispatcher starts a remote execution processon each node specified in a special configuration file. If the user has provided passwords in this file, then the processes will be started transparently. If the passwords are omitted, then the dispatcher prompts for them from standard input.

Files and Filenames

Clustor does not assume a shared file system, thus files must be transported between the root machine and the computational nodes as required. This is usually acheived by a special command which copies files between machines (See "COPY" on page 31.).

When clustor starts a job on a remote system it ceates a unique place in the remote file system, and all relative file names are interpretted relative to that position. This scheme makes it possible to run the job on the same machine (or in the same shared files system) concurrently without file names conflicting. For example, each job can write to a file called "output", which is a unique file on each node.

When files are copied back from a node, the file name chosen on the root system must be modified to include a unique identifier. This can be doen by using a special parameter (called $nnn) which is expanded for each unique job. Thus the following command, copies a set of files, all called "output" back to the root machine, and assigns a unique name at the root.

```
COPY node:output root:output.$nnn
```

If files are copied to the computational node before the job has started (for example, at Node Startup time) then they are in turn copied into the unique space as jobs start on that node. This means that it is possible to copy in a set of files which must be shared by all executions of an application, once, when the connection is made with the computational node. As each job starts, they are copied into the unique position in the file space. This scheme is more efficient than copying the files when the job starts because they are only moved across the network once per computational node.

CHAPTER 4 *Writing Plans*

In this chapter the syntax and semantics of the various Clustor commands are discussed.

Structure of Plans

Plans contain two main types of information, namely, a description of the various parameters and instructions on how to run the application and perform the file transfer. These are partitioned into two main groups of commands, *parameter* statements and *tasks*.

The Parameter Statement

The parameter statement specifies the name, display properties and type of the parameter as well as the method of generation of parameter values. It has the following syntax:

```
parameter   <name> [ <label> ] [ <display> ]
            <type> [ <domain> ] [ <default> ];
```

where:

```
<name>      a sequence of non-space printable characters
<label>     label "character string"
<display>   displayon | displayoff
<type>      integer | float | logical | text
<default>   default <value>
<value>     a sequence of non-space printable characters|
            "character string"
<domain>    <empty> [ default <value> ] |
            select anyof <vlist> [ default <vlist> ] |
            select oneof <vlist> [ default <value> ] |
            range [ from <value> ] [ to <value> ] [ points
            <value> ] |
            range [ from <value> ] [ to <value> ] [ step <value>
            ] |
            random [ from <value> ] [ to <value> ] [ points
            <value> ]
            compute <expression>
<vlist>     <value> <vlist> | <empty>
```

The `<name>`, specified in the parameter statement is used by all programs to denote the parameter. The optional parameter label, `<label>`, is displayed by the graphical user interface to label the parameter. If the label is not specified, the parameter `<name>` is used on the graphical user interface.

The optional display argument, `<display>` determines whether the parameter is shown by the graphical user interface or not. If **`displayon`** is specified, the parameter is shown. If **`displayoff`** is specified, the parameter is not shown. The default value is **`displayon`**.

The parameter type, `<type>`, specifies the type of the parameter value. The type can be either `integer`, `floating point` or `text`. The type is used to verify user input. On output, all values are treated as text.

The optional parameter domain, <domain>, determines the method, by which the user can specify multiple values for the parameter. If the domain is not specified, a single value is produced for this job. The method of specifying the domain can be one of the following:

- **select anyof**: the user can select any combination of the values in the value list, <value list>
- **select oneof**: the user can select one of the values in the value list, <value list>
- **range**: the user can specify the number of values between the specified range. The number of values, the lower bound and the upper bound can be all user specified. Values will be generated uniformly between the lower and the upper bound. Alternatively, the user can specify an increment value. In this case, values will be generated starting with the lower bound and incremented until the upper bound is exceeded.
- **random**: random numbers between the lower and the upper bound will be generated. The user can specify the lower bound, the upper bound and the method by which random values are generated. If the generate option is specified and its parameter is a nonzero, positive integer, random numbers will be generated before the generation of jobs. The number of these random numbers will be determined by the value of the generate option parameter. In this case, this parameter influences the number of jobs generated and each random value can be used by multiple jobs. If no generate option is specified or its parameter is 0, random numbers will be generated independently for each job during the generation of jobs. In this case, this parameter will not increase the number of jobs and each random value will be used by one job only. Default values can be specified for a single value or for select domains.
- **compute**: This statement allows the user to compute an expression based on other parameter values. Compute parameters are not shown in the generator user interface. They behave as internal variables and can be used

anywhere a normal parameter can be used. Expresssions are standard arithemtic expressions using the +, -, * and / operators. The % (mod) operator is also available for integer parameters. Functions which are supported include ln - natural logarithm, log10 - log base 10, exp - power of e, exp10 - power of 10, sqrt - square root. These can be used to generate logarithmic, exponential and square root distributions.

Examples of parameter statements

```
parameter  var1 label "variable1" integer
           select anyof 1 2 3 4 5 6 7 8 9 0 default 1 5 8;
parameter  var2 integer
           select oneof 1 2 3 4 5 6 7 8 -9 0 default 1;
parameter  var3 text label "input text"
           select anyof "1 2 3" "a b c" "x y z"
           default "1 2 3";
parameter  var4 integer;
parameter  var5 integer range from 0 B 100 points 3;
parameter  var6 displayoff float
           random B -1.0 to 1.0 points 6;
parameter  var7 integer random from 1 to 6;
parameter  var8 float compute sqrt(var1)*sqrt(var1);
```

Tasks

Tasks specify the actions which are executed during a job run. Each task description starts with the task statement, followed by user statements, and ends with the endtask statement:

```
task <name> <user statement list>
endtask
```

where:

```
<user statement list> is      <user statement> <newline>
                              <user statement list> |<empty>
```

Task Names

Several task names are predefined by the system. These are:

- “main”,
- “rootstart”,
- “rootfinish”,

- “nodestart” and
- “nodefinish”.

At the beginning of a run, the “rootstart” task is executed. At the end of a run, the “rootfinish” task is executed. The “noderstart” task is executed on each computational node, before a job is run on that server. The “nodefinish” task is executed on each computational node after all the jobs have completed. The “main” task specifies the execution of a job. In addition to the system defined tasks, the user can specify any number of user defined tasks.

System Defined Parameters

Several parameters are predefined by the system. These are:

- “nnn”
- “node”,
- “run”.

During the execution, these parameters are replaced by the job, node and run identifiers, respectively.

Locators

Locators are machine addresses which can be used in script commands and file descriptions. For script commands, they specify the machine to execute the command. For files, they specify the location of the file. Locators can be any of the following:

- “root”,
- “node”,
- “local”,
- “remote” or
- <network address>.

“root” and “node” are absolute addresses, specifying the root and the node machine for each job. “local” and “remote” are relative addresses. On the root machine, “local” is the root machine and “remote” is the node machine. On the node machine, the meaning is reversed. <network address> can the name of any machine on the Internet. If a locator is omitted, its default value is "local".

Naming the configuration file

By default, Clustor looks for a node configuration file called clustor.nodes in the working directory. It is possible to use a different file name using the configure command, as follows:

```
configure "<filename>"
```

where `<filename>` is a standard Unix file name.

Include files

It is possible to include other files in a plan using the include command, as follows:

```
include "<filename>"
```

where `<filename>` is a standard Unix file name.

Task Commands

The following commands are currently supported:

```
append
copy
delete
execute
substitute
```

Each command can include a locator, which describes the machine to execute the command, for example `server:copy`.

APPEND

```
append <source file> <destination file>
```

The command appends a source file to a target file. If the target file does not exist, it is created.

COPY

```
copy <source file> <destination file>
```

The command copies a source file to a target file. If the target file exists, it is cleared first. The COPY command allows the `<source file>` and `<destination file>` to contain wild card characters, namely, *, ? and

[]. These have the same interpretation as in Unix file names. The `<destination file>` can also be a directory name, in which case all files are copied into the target directory.

DELETE

```
delete <file>
```

The command deletes a file or directory.

EXECUTE

```
execute <user command>
```

This command executes an arbitrary user command.

SUBSTITUTE

```
substitute <source file> <target file>
```

The command substitutes all parameter references in a source file with parameter values and produces a target file. If the target file exists, it is cleared first. The parameter is named in the source file using the following syntax:

$<parameter>

where <parameter> is the name of the parameter in the plan.

CHAPTER 5

Machine Configuration

Management

Clustor hides most of the management of the distributed platforms from the user. It uses a robust mechanism to connect with remote nodes, automatically starting severs when required. Unlike other systems, it is not necessary to install any system management code with special privileges on a local or remote system. The only assumptions that Clustor makes about remote systems is that it is possible to connect to them by the industry standard `telnet` tool and that each user has installed the Clustor server software. In this way, Clustor does not compromise the security of any system over and above that of the `telnet` utility.

Configuration

When Clustor is started it looks for a configuration file called clustor.nodes in the current directory. This file contains a list of machines which can be used, plus login information. The user name, and an optional password is stored for each machine. If a user does not wish to store the passwords in the file, they can be omitted, in which case the dispatcher will prompt for the password every time it connects to the remote system.

Each line of clustor.nodes has the following format:

clustor.nodes

full machine name	username	password
full machine name	username	password

Comments are denoted by a '#' character at the beginning of a line.

If a number of machines have the same password, it is possible to indicate this by naming the machine in the password field, preceded by a '$' sign. For example:

clustor.nodes

mach1.university.edu	username	
mach2.university.edu	username	$mach1.university.edu
mach3.university.edu	username	$mach1.university.edu
mach4.university.edu	username	

Here, mach1.university.edu, mach2.university.edu and mach3.university.edu all share the same password, even though it has not been entered into the file. Thus, when Clustor prompts for the password for mach1.university.edu, it will use the same password for mach2.university.edu and mach3.university.edu without having to ask again. However, the system will prompt for a password for mach4.university.edu.

Security

Remote execution servers are run under the username given in the config file, and thus Clustor assumes that the user already has an account on the machine. It also assumes that the files required for running the application are available, or are sent as part of the user script.

CHAPTER 6 System Management and Installation

What do I have to do to install Clustor?

Clustor can be down loaded from the Active Tools Web site. To obtain a copy, perform the following:

- Download the compressed tar file for the required binary
- Uncompress the file using `zcat` or `uncompress`
- Untar the binaries into a directory which is on your search path
- Obtain a license from the Web form and place this in the executable path
- Prepare a `clustor.nodes` file
- Download a copy of the user manual.