A Versatile Beowulf Task Distribution Application

Clif Flynt
Noumena Corporation
`www.noucorp.com`
`clif@noucorp.com`

1 Abstract

Running faster has been the holy grail of computing since the days of the abacus. The first thing a programmer hears after "does it work" is "can you make it run faster?"

In the early days of computing, the best way to make a program run faster was to find a better algorithm or optimize the commands. After some 50 years of study, most of the better algorithms have been discovered and put into well optimized libraries.

The next way to make an application run faster is to split it into smaller applications and run them on multiple processors. Modern CPU chips do this to some extent and modern Graphics Processing Units do it to a greater extent.

Problems that can be optimized for parallel computing range from fine-grained applications in which the behavior of one thread is influenced by the computations of other threads to coarse-grained application sets that are totally separated from each other.

Fine-grained applications in which one thread influences another require tools like PVM (Parallel Virtual Machines) or CPS (Concurrent Processing System), specialized hardware with high-speed interprocess communication (often shared memory) and generally involve instrumenting the code or writing a special application to perform the processing.

In medium and coarse grained applications the behavior of one processing thread does not influence the behavior of other threads. These problems are much more approachable with simple hardware and relatively trivial care in the architecture of the processing applications.

Examples of medium-grain parallel tasks might be performing image processing in a set of strips and then reassembling the strips into a complete image, or generating a mandelbrot set as a collection of areas that are then assembled into a mandelbrot image. These individual tasks may run at different speeds depending on the resource used and the complexity of the task. When all of the data is available, a final result can be created.

Examples of coarse grained parallel tasks include running multiple simulations with different sets of data, calculating a fitness of solutions for a genetic algorithm, or performing the same analysis on multiple datasets. These applications are completely independent of each other, although the results sets may be combined for later analysis. This level of parallelization is the basis for the "Seti@home" and "folding@home" projects. The application requires a small amount of data and a large amount of independent data processing.

Beowulf clusters, sets of computer nodes with slow (ethernet-speed) interprocess communications are suitable for applications with medium or coarse-grain parallelization which require large ratio of processing time to inter-system communication.

The concept of a Beowolf cluster covers a large range of autonomous processing units from dedicated, diskless-compute nodes to standalone workstations that aren't being fully utilized. The techniques for distributing the tasks can range from direct memory access to scp/ssh interactions.

It is relatively simple to distribute a set of tasks across multiple computing resources. For a small number of nodes and tasks, the tasks can be distributed and hand-started. For a single application and a well controlled set of nodes, a simple shell or Tcl script can be used to start applications as necessary.

Creating a special-purpose control application for each application that needs to be distributed is costly in terms of human time and reduces the set of applications that can profit by being distributed. There is a need for a generic application that can be extended to handle multiple types of tasks and multiple styles of clusters.

The mythical Beowolf met Wulfgar when he first came to Heorot. Wulfgar escorted Beowulf to the king and thus provided him with his first task. Wulfgar was the person who connected a resource (Beowulf) with a task (Grendel).

This `wulfgar` is a Tcl/Tk application and framework for creating and distributing tasks across a set of resources (compute nodes). It can be used from a command line or a GUI. It can be extended for new types of projects by defining a new class and can be extended to control different Beowolf architectures by adding external applications to interact with nodes in different style of cluster.

While it would have been great fun to continue the naming motif and have `wulfgar` distribute quests across a network of castles, the references to ancient Geats and monsters ends with the application name.

2 Overview

The `wulfgar` application distributes a set of jobs among a set of computing nodes. The jobs will be run one-at-a-time on the nodes and the results will be collected into a defined location. One job is distinguished from other jobs by it's command line arguments. The arguments may be simple values (like `-x 1 -y 3`) or the name of a configuration file.

The jobs are grouped in a *`project`*. The project defines the executable to be used and how individual jobs are created from that project. The current version of `wulfgar` can create jobs using a single numeric loop, two nested loops, or from a set of configuration files. New project types can be created by adding a new classes with a custom constructor that creates jobs for this style of project.

The computing nodes are grouped into a ***nodeSet***. A nodeset is a collection of remote nodes which share a common access method (`ssh, rsh, shell`, shared memory, etc.) New types of nodeSets can be created by writing new access scripts.

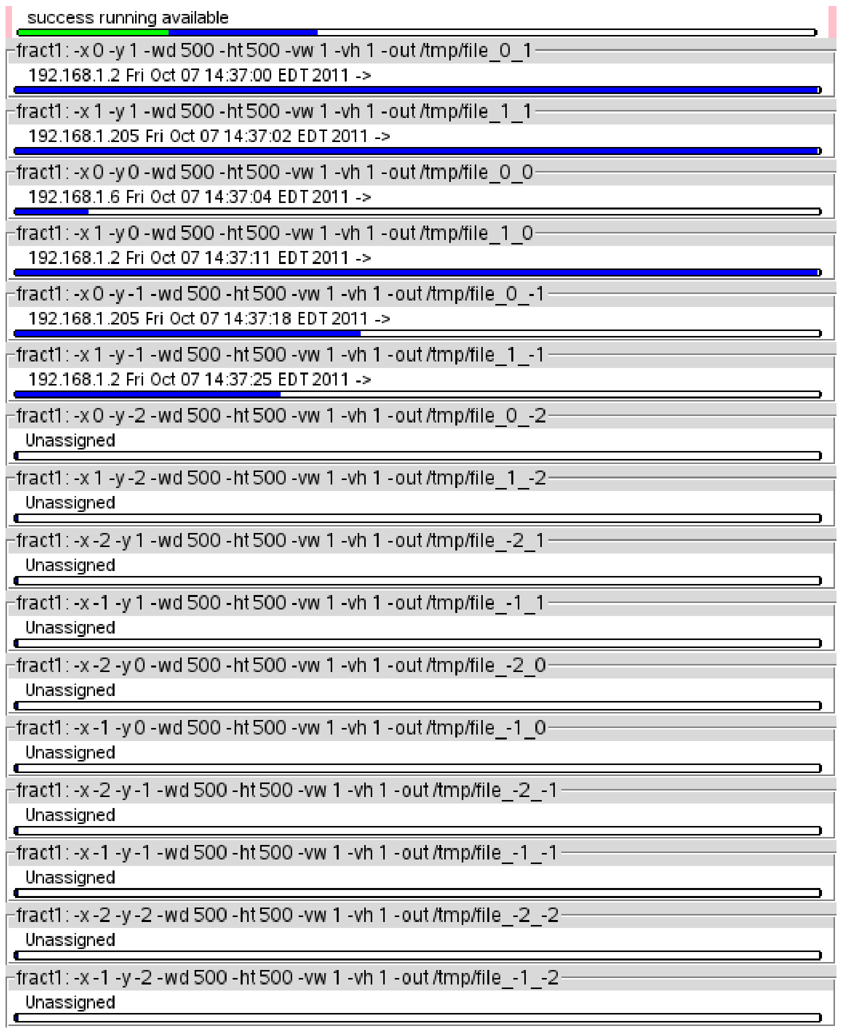
The `wulfgar` application is written using TclOO and a Model-View-Controller design paradigm. The base classes control defining and distributing jobs and nodes and collecting the results. The GUI uses inheritance, mixins, the `info` command and the `trace variable` command to examine and attach itself to the controller elements of the application. This allows `wulfgar` to be run either from a command line or script or by interacting with a user via a GUI.

When running in a GUI mode, a running task resembles the image below. This shows a set of 16 tasks assigned to 3 nodes on an internal network. The jobs are distinguished by the `-x`, `-y` and `-out` command line values. The `-vw, -vh, -wd` and `-ht` arguments are the same for each job.

The top line shows the progress on the project - the collection of tasks. There are jobs in 3 states, success, running, and available. The three colors shown in the completion bar show the relative number of tasks in each state.

Each of the lines below shows the status of that task, either that it is complete, the percentage of completion, or that it is available and unassigned to a compute node.

Tasks that have been assigned show the node that they are running on and when they started. A completed task also displays the end time.



```
success running available
fract1: -x 0 -y 1 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_0_1
  192.168.1.2 Fri Oct 07 14:37:00 EDT 2011 ->
fract1: -x 1 -y 1 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_1_1
  192.168.1.205 Fri Oct 07 14:37:02 EDT 2011 ->
fract1: -x 0 -y 0 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_0_0
  192.168.1.6 Fri Oct 07 14:37:04 EDT 2011 ->
fract1: -x 1 -y 0 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_1_0
  192.168.1.2 Fri Oct 07 14:37:11 EDT 2011 ->
fract1: -x 0 -y -1 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_0_-1
  192.168.1.205 Fri Oct 07 14:37:18 EDT 2011 ->
fract1: -x 1 -y -1 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_1_-1
  192.168.1.2 Fri Oct 07 14:37:25 EDT 2011 ->
fract1: -x 0 -y -2 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_0_-2
  Unassigned
fract1: -x 1 -y -2 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_1_-2
  Unassigned
fract1: -x -2 -y 1 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_-2_1
  Unassigned
fract1: -x -1 -y 1 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_-1_1
  Unassigned
fract1: -x -2 -y 0 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_-2_0
  Unassigned
fract1: -x -1 -y 0 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_-1_0
  Unassigned
fract1: -x -2 -y -1 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_-2_-1
  Unassigned
fract1: -x -1 -y -1 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_-1_-1
  Unassigned
fract1: -x -2 -y -2 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_-2_-2
  Unassigned
fract1: -x -1 -y -2 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/file_-1_-2
  Unassigned
```

3 Internals

`Wulfgar` views the world as collections of static and ephemeral elements. The static elements are instantiated as classes which are reflected into a database (using `tdbc::Sqlite`) The ephemeral entities are implemented as memory resident classes which are created when needed and are discarded when `wulfgar` terminates.

The static elements are collections of jobs and resources. A job's non-volatile state includes the executable to invoke and the arguments to be used with the executable. A job has a volatile characteristic of whether it has been run and the final status of the run. A resource (referred to as a node) has a non-volatile state that includes the IP address, access port and a volatile attributes of online/offline status.

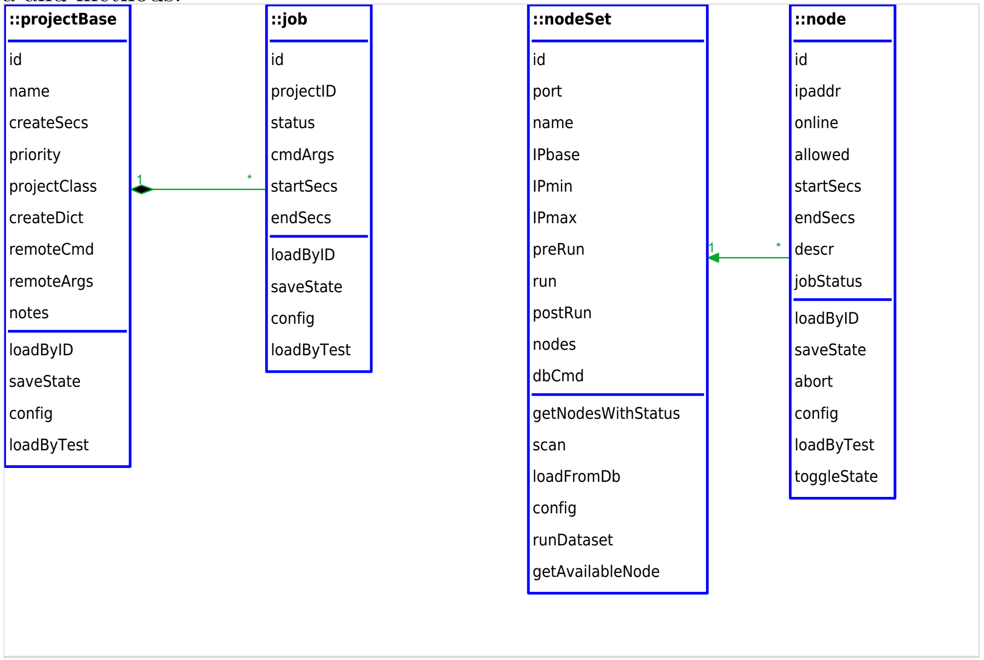
The ephemeral entities are the set of jobs and resources currently in use. These are grouped as a job and a resource when the job starts being executed.

A job is a single run of an executable program and a set of data. Jobs are collected into a `project`, which is a collection of jobs that use a common executable on the processing resource.

Compute resources are referred to as `nodes`, and are grouped into `nodeSets` A `nodeSet` is a collection of nodes that share a common access manner such as ssh, rsh, etc.

The `project`, `job`, `nodeSet` and `node` are relatively static entities. They have a state (available, running, completed) that is modified but otherwise they exist from the time they are created until a real-world project is complete and the database is retired.

The `project`, `job` `nodeSet` and `node` classes are shown below with their data and methods.



Each of the classes has an `id` variable. This is used to reflect the data to the database but is not otherwise used by the runtime system.

Each class has a `config` method. The `config` method modifies the contents of a variable and updates the modified data in the database.

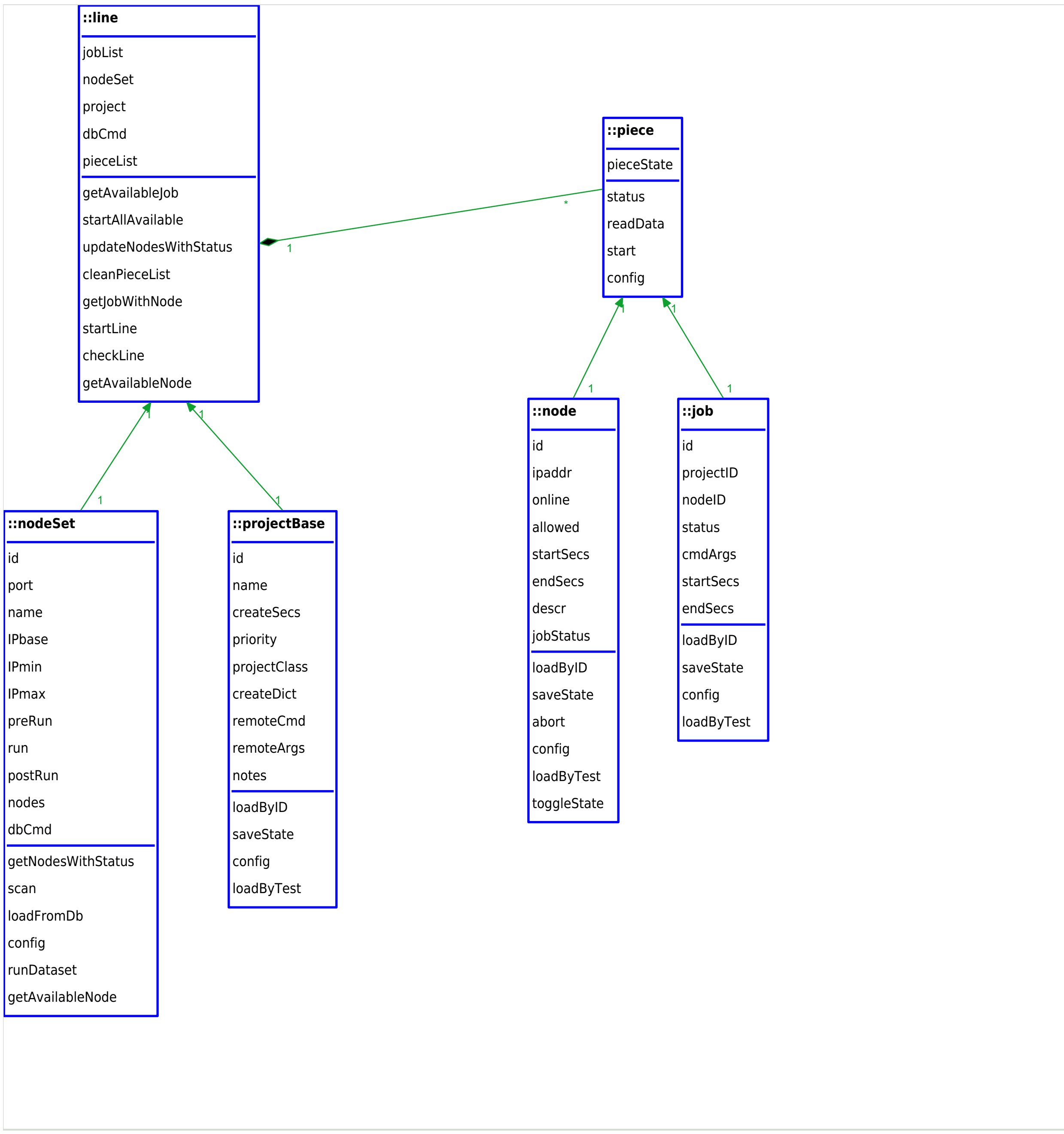
The relationship between a job and a node that it is running on is ephemeral. These items are created as required and destroyed after they have been used.

The ephemeral classes in `wulfgar` are the `line` and the `piece`. The line is named for a manufacturing line. It contains a nodeSet and a project. The `piece` (for piecework) is a class with a single job and a node to run the job on.

A line has an active project and an associated nodeset. When a line is running, it creates pieces by selecting an available job and node and creating a piece object that contains the job and node. Once a `piece` has been started, it interacts with the node using the pre-run, run and post-run external applications defined by the `nodeSet`.

A `line` object contains the name of a `project` object and a `nodeSet` object and a list of the `piece` objects that it has created from the jobs and nodes in the `project` and `nodeSet`.

The next image shows the relationships between line, project, nodeset, piece, job and node.



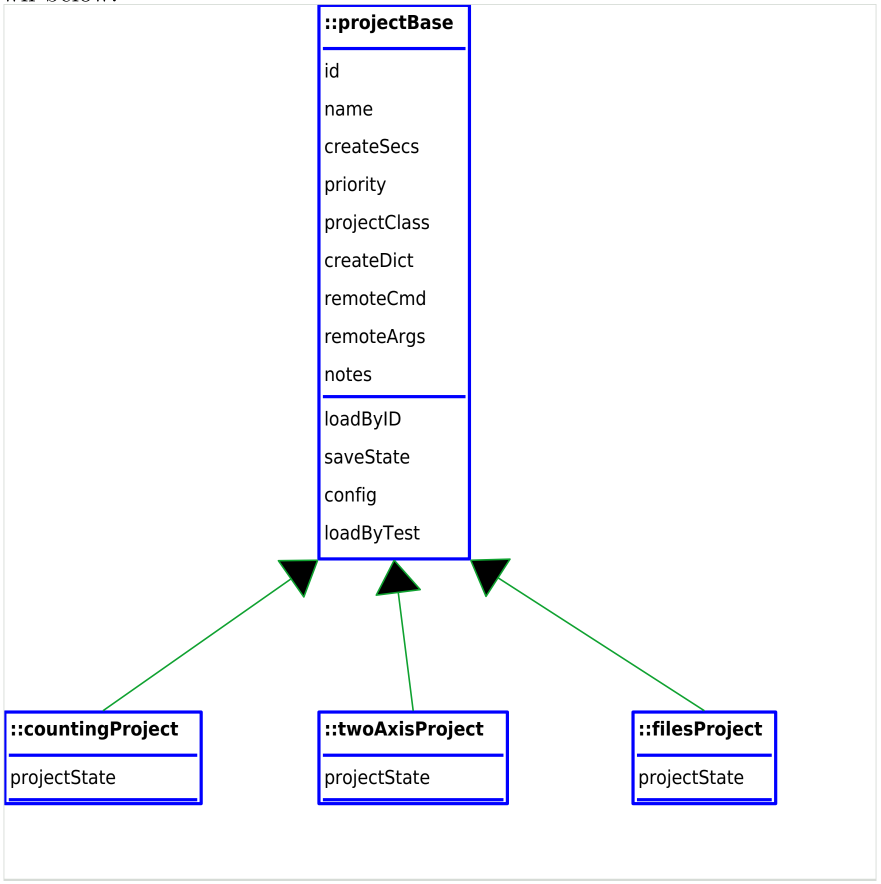
The project, job, nodeset and node objects are all reflected into an SQL database, allowing processing to be stopped and restarted as necessary at the cost of losing the work done by tasks that are currently running on remote nodes.

4 Creating tasks and jobs

When a project is created the constructor also creates a set of jobs. Different types of projects use different methods of creating tasks. As examples, tasks to model the behavior of an engine running different grades of fuel could be created by iterating through a single loop of octane ratings. A set of tasks to fill areas of a Mandelbrot set could be created with nested loops iterating over an X/Y area to define rectangles to be computed. A set of tasks to model the behavior of a complex system could be generated by iterating through a set of configuration files generated by external applications.

Support for multiple styles of creating tasks is implemented within `wulfgar` by deriving classes with type-specific constructors to create the associated jobs from a standard base class (`projectBase`). New mechanisms for creating jobs can be added by creating new project classes

The inheritance relationship between the parent class (`projectBase`) and the derived classes `countingProject`, `twoAxisProject` and `filesProject` is shown below.



The important attributes of a project are:

`name`	Used to identify this project to a human.
`priority`	Used to schedule this project when multiple projects are active.
`createDict`	A set of key/values that are used to create jobs for this project.
`remoteCmd`	The command to run on a remote system.
`remoteArgs`	A set of patterns to use to create the command arguments for the remote task. The remoteArgs string may include tcl variables or commands which will be substituted using the `subst` command when the job object is created.

A project to generate rectangular areas of a mandelbrot set resembles this:

`name`	`mandelbrot`
`priority`	`1`
`createDict`	`startX -2 startY -2 endX 2 endY 2 incrX 1 incrY 1`
`remoteCmd`	`fractal.tcl`
`remoteArgs`	`-x $x -y $y -w 25 -h 25 -vh 1 -vw 1 -out mdl_$x_$y`

The `createDict` is used to initialize the nested loops. The loops variables are `x` and `y`, which are used (with Tcl's `subst` command) to populate the arguments to the jobs.

A job's attributes include

`projectID`	References a project's ID in the databae.
`status`	Describes the job's status: available, success, fail, abort.
`cmdArgs`	The command line arguments for this job.

An individual job within the `mandelbrot` project would resemble this:

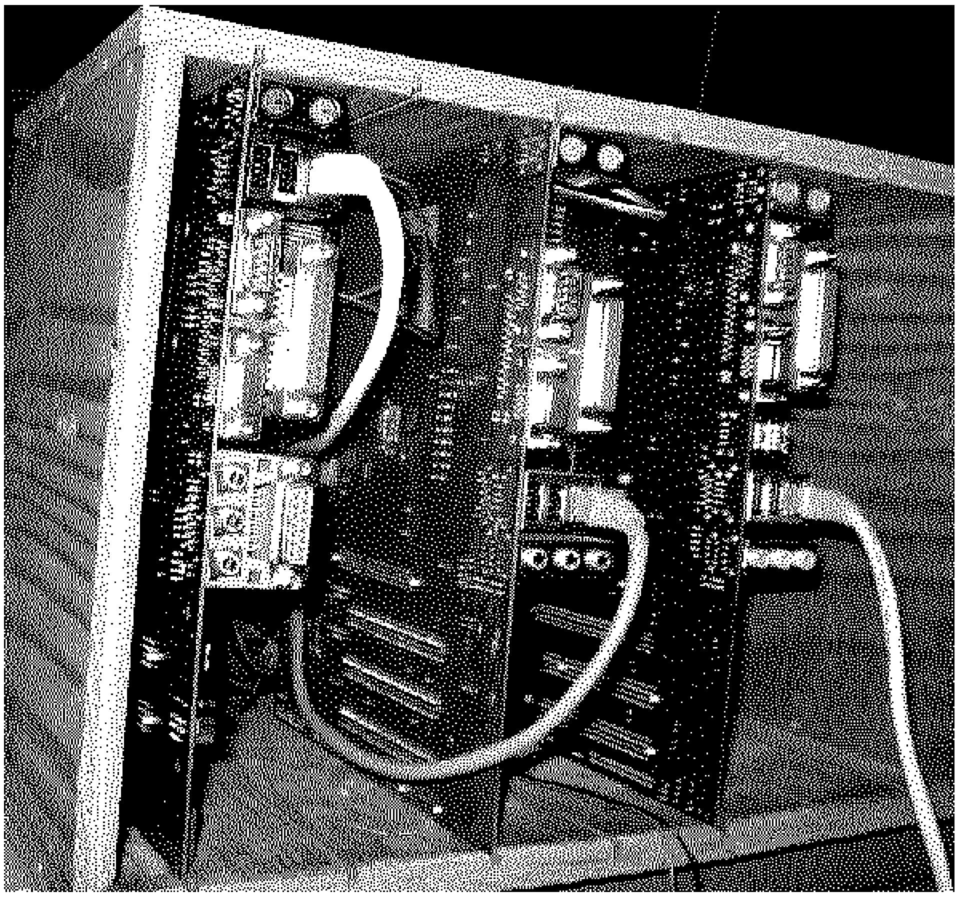
`status`	available
`cmdArgs`	`-x 0 -y 1 -wd 500 -ht 500 -vw 1 -vh 1 -out /tmp/mdl_0_1`

5 Distributing Jobs

A classic beowolf cluster of stand-alone processors distributes jobs using `ssh/scp` or `rsh/rcp` across a network. Setting up a compute cluster using workstations is fairly cheap (by supercomputing standards) and easy. However, racks of cabinets use a lot of space and running and maintaining dozens of disk-drive based systems can chew up a lot of human time.

An economical compute cluster can be created with a set of diskless motherboards attached via an on-board network adapter to a server. The diskless nodes can be booted using the PXE environment and data transfers can be done using a NFS shared partition, the traditional `ssh` or the computationally cheaper `shell (port 514)` protocol.

A functional compute cluster can be created from castoff motherboards thumb-tacked to a cubicle wall, or as assembled into a box as crudely as shown below:



The `wulfgar` application can be extended to different connection architectures with external applications that are `exec'd` by `wulfgar` to transfer the client application to the target system, execute it and collect results.

These applications will receive a set of values defined for the nodeset (IP address, port, userID, password, etc) and per-job values which they must parse for more details.

The `Expect` extension is a very useful tool for this sort of machine control and is used in the external applications provided with `wulfgar`.

The example below is a sample of a `post_scp.tcl` application which copies a result file from the remote system to the local results folder. It receives the user and password from the values in the nodeSet's `postRun` attribute, and other values (`-out`) from the values assigned to the job's `cmdArgs` attribute.

```
#!/opt/ActiveTcl-8.6/bin/tclsh8.6
lappend auto_path .
package require expectTools

exp_internal 0
```

```
log_user 0

if {[llength $argv] < 4} {
  puts {post_scp.tcl -local localFolder -user loginID -pwd pwd }
  puts {From Run: post_scp.tcl -path remoteFile \
      -ip [$n config -ipaddr]}
  exit
}
puts "[llength $argv] ..$argv.."
array set av $argv

if {![info exists av(-path)]} {
  set av(-path) $av(-out)
}

if {![info exists av(-local)]} {
  set av(-local) .
}

spawn scp $av(-user)@$av(-ip):/$av(-path) $av(-local)
dialog assword $av(-pwd)
dialog 100% ""
```

The remote application on the compute node can be a single executable, or a script that invokes several cooperating applications.

6 Adding a GUI

Since `wulfgar` is an expandable, adaptable application, the GUI code needs to self-constructing and as independent of the thinking parts of the application as possible.

This is accomplished in `wulfgar` with some naming conventions and by using TclOO inheritance and mixins, Tcl's introspection (particularly `info class`) and trace facilities.

One convention is that the `config` command behaves like the Tk *`widgetName`* `configure` command in that it returns a list of keys and values when it is invoked with no arguments. This allows a procedure to easily retrieve a list of the attributes in an object's state array that can be assigned values.

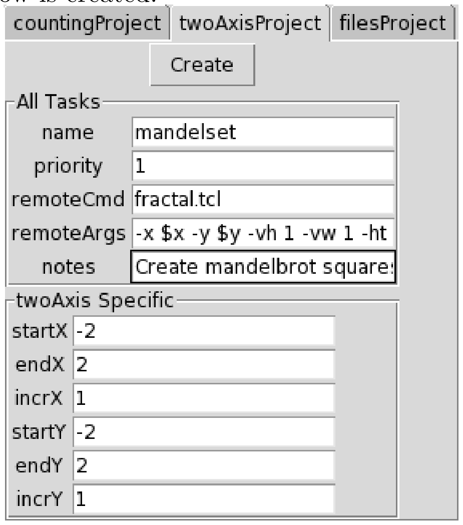
This convention allows the GUIs that create an object to query the class for values to be used in defining the object.

The `info class` command provides access to the class state of an application. This can be used to determine how many classes are derived from another class and dynamically construct a GUI that reflects the available functionality.

The next code snippet demonstrates using the `info class subclass` command to find the classes for the specific project types and create a separate tab in the tab notebook for each class. The `configure` command is used to populate the fields with the class specific attributes and their initial values.

```
set nb [ttk::notebook $fr.nb]

foreach nm [info class subclass ::projectBase] {
  set nm [string trim $nm :]

  $nb add [frame $nb.f_$nm] -text $nm
  set f2 $nb.f_$nm

  set f3 [labelframe $f2.fs \
      -text "[string range $nm 0 end-7] Specific"]
  grid $f3 -sticky news

  foreach {k v} [$tmp config -createDict] {
    label $f3.l_$k -text $k
    entry $f3.e_$k -textvariable ::GUI($nm,cD,$k)
    set ::GUI($nm,cD,$k) $v
    grid $f3.l_$k $f3.e_$k
  }
}
```

When combined with the rest of the project GUI code an GUI like the image below is created.



name	mandelset
priority	1
remoteCmd	fractal.tcl
remoteArgs	-x $x -y $y -vh 1 -vw 1 -ht
notes	Create mandelbrot square:

startX	-2
endX	2
incrX	1
startY	-2
endY	2
incrY	1

Another convention is that the external nodeState applications are named using a pattern (so that `glob` can identify them) and must return a list of arguments when invoked with no arguments. This allows the GUI for creating a nodeset to automatically expand when new nodesets are created.

A common design pattern is for a class with GUI methods to be inherited from a compute model class. This pattern is used to define the lineGui and nodeSetGui classes.

While a `line` is running, the line object creates new piece objects as it needs them. It's not convenient for a `lineGui` object to create the `pieceGui` objects, since it's not involved in the `piece` creation.

The `line` and `piece` classes interact with each other, and those interactions need to be displayed in a GUI without touching the code that controls the interactions.

The `line` class maintains a list of pieces that have been created.

The `lineGui` class inherits from the `line` class and places a `trace` on the `lineState(pieceList)` variable. This allows the lineStateGui to be updated whenever the `line` object adds or removes a piece.

As a remote task runs it should report a fraction complete message at intervals. This message is received by the `piece` object that is controlling the task and saved in the `pieceState(complete)` attribute.

Like the `lineGui`, the `pieceGui` class uses a trace on the `pieceState(complete)` variable to update the completion bar.

7 Conclusion

Controlling disparate tasks on remote systems is a solvable, but non-trivial problem. The difficulties in a generic solution are the different methods of communicating with remote nodes and automating the creation of tasks.

The `wulfgar` solution is to provide a common framework for controlling nodes and tasks, with relatively small bits of glue in the form of customized classes and external applications to allow for per-application and/or per-site customization.

Tcl's ability to exec remote tasks coupled with `expect`'s ability to interact with remote systems enables a user to tweak the control applications to match their system.

The `TclOO` support for both inheritance and mixins and the ability to load new code at runtime and introspect to find what classes can be created provides a powerful environment for customizing task creation and linking tasks with resources.

The `trace` and `mixin` facilities are a powerful tool to create a framework in which the GUI and calculation code can interact with each other without being intermingled.