Owen™ — the Inferno Compute Grid

Roger Peppe

Robert Owen (1771-1858) was a British entrepeneur and philanthropist amongst whose achievements was the formation of an *equitable labour exchange system*: exchanging hours of work for labour notes between cooperative societies. The Inferno Compute Grid, named in his honour, brings something of the same approach to the distribution of computing workload.

The basic notion is that any machine acting as a compute node can approach the central hub of the system, also known as the *scheduler*, announce that it is willing to take on certain kinds of work, and wait to be given some work to do. Other nodes approach the scheduler with work to do. The scheduler matches workers and tasks, and also makes sure that idle and erroneous workers do not get in the way of the smooth running of the system.

The Owen system is implemented using the Inferno system, which was designed for the implementation of distributed systems. Inferno is lightweight and portable, running applications unchanged under Microsoft Windows, Linux, FreeBSD, MacOS X and others. It can also run as a native OS on many species of hardware. Although applications inside Inferno are written in the Inferno-specific language Limbo, interfacing to native application code from within Inferno is straightforward. In particular, there is a way of starting and controlling existing host applications from Inferno.

System overview

An Inferno computational grid system has three components:

- Scheduler
  The scheduler is the labour exchange: it matches job tasks with appropriate workers, mediates data transfer, and checks that the tasks have been completed to a satisfactory level before marking them done.
- Workers
  Workers are the nodes (clients) that actually perform the work for the grid. The idea is that a worker goes to the labour exchange (the server), announces the kind of work that it is willing to perform, and waits until it is given some.
- Controllers
  Controllers also connect to the labour exchange, but instead of asking for work, they post jobs of work to do. Each job consists of several tasks, each to be performed by a worker node (some or all of these tasks might require a specialised worker node to perform them).

The fundamental work component used by Owen is the *job*, representing a collection of work to be done. A job is split into a set of smaller independent computations, called *tasks*, each of which is handed to a worker to do. Because tasks are independent, an arbitrary subset of them can run on many processors concurrently. For example, a cryptographer might create a job to test a large set of keys against some encrypted data; the key space would be split into chunks, each tested by a separate task, allowing many chunks to be tested at once, each by a task on a different worker node.

Given enough nodes and a large enough network, something is bound to fail sooner or later during the course of a job, especially when a single task, let alone an entire job, takes significant time or processes a large amount of data. Some systems provide elaborate schemes for automatic checkpointing and process migration. This system instead relies on redundancy. The division of a job into tasks can not only increase concurrency, the system can rely on it to improve reliability. If there is a node or network failure, the scheduler can automatically reschedule a task on another node. In other words, the system ensures that each task will execute *at least once*. It follows, however, that a task should be small enough that there is a good chance that an average node will be able to complete it within its average up-time.

A *task generator* is responsible for splitting a job into separate tasks. Starting an Owen job entails starting a task generator with some job-specific parameters that allow it to create appropriate tasks. In the code cracking example, the parameters might describe the size of the key space, specify the encryption algorithm, and give the data to be decoded; the task generator might then generate tasks form sections of the code space until one task announces that the code has been cracked, whereupon the job would be declared complete. The Owen scheduler is responsible for resilience and error-recovery: it will ensure that all tasks are completed, regardless of misbehaving clients, network outages, device failures, and other such faults. Task generators are not trivial to write, and how to write one is beyond the scope of this document, but Owen provides several general-purpose task generators, each ready-made to handle some common class of job. One of them, known as `filesplit`, is given a file and creates tasks each of which processes some specified subset of the file. Another, known simply as `job`, is designed for a wider range of jobs that have a known number of tasks, each requiring some input in the form of file data and parameter values, possibly differing per task. Examples are given below.

1. Grid organisation

The first thing to decide is where the scheduler should reside. It needs to run on a generally accessible machine; it listens on a network port for connections, so that port on that machine must be accessible to all machines that are to take part in the Owen grid. The scheduler installation is essentially a complete Inferno installation with the addition of the Owen software. Instructions for its installation are given in section NNN.

After installing the scheduler, you will need to install the software on each worker (also known as 'client') node. The worker software consists of a stripped down Inferno installation, holding just enough software to run an Owen client, and to run the Owen monitor and submit new jobs (ie, any Owen worker can potentially act as a Controller node too).

For a small number of nodes, the easiest thing is probably to install the software individually on each of those nodes (Instructions for its installation are in section NNN).

When installing on larger numbers of nodes, there are various possible approaches, depending on the local setup. The first thing to do is to make a template installation, containing all the files as they will be needed on each client. Just following the above-mentioned instructions for client installation is usually good enough. Note that you need to consider which tasks a worker will deign to perform before actually deploying the installation. This is a significant security consideration.

You then need to copy the installation to all the other client nodes. The *VN Grid Client Replicator* can often be used to do this on Windows machines (see here for documentation).

On non-Windows networks, the installation will depend on your local setup. The usual procedure is to copy the installation tree onto a filesystem on each client, and to arrange for the client software to be started automatically at boot-time. When a set of nodes share a common filesystem, it is important that at least the directory `/grid/slave` is unique for each running installation, as this is used to hold scratch data, and several problems will result if several nodes use the same directory at once.

In the examples, we assume a local-area network of some Windows or Unix PCs (or a mixture), firewalled off from the wider Internet. This is possibly the simplest kind of installation; we assume that the network is trusted, and that the client machines are prepared to run anything that the server asks them to.

There are many ways that you can put the compute grid together. The software can be configured as required for each application, and indeed many of our production grids have done so.

The first two examples, however, keep customisation to a minimum, because they show methods that work well for several common cases.

2. Example: set up and test the grid

1 **Choose a server machine to run the scheduler.**
The right machine to choose will have a high up-time, preferably a dedicated machine. It is usually a server, but could be someone's desktop machine, if that is suitable. In any case, the machine has an internal firewall, as some Windows machines do, you will need to enable incoming connections on at least one port. Conventionally, port 6678 is used, named `infsched` in Inferno. Inferno network addresses are expressed as textual strings containing three parts, separated by exclamation-mark (`!`) characters. The first part names the type of networkor protocol to use (for instance, ''`tcp`''); the second gives the name or address of the machine on that network, for instance a DNS name such as ''`gridserver.somewhere.com`'' or an IP address such as ''`200.1.1.94`''; and the third part gives the name of the service or its port number, for instance ''`infsched`'' or ''`6678`''. Thus the full address of an Inferno scheduler might look like

```
tcp!gridserver.somewhere.com!infsched
```

You can use `net` instead of `tcp` to use any available protocol or network to reach the host.

2 **Install the software**
Follow the given instructions to install the scheduler software on the server and then the client software on each machine that is to be a worker on the grid, making sure that the server address given as the value of the `schedaddr` attribute in configuration file `/grid/slave/config` is set to point to the correct server.

3 **Allow arbitrary execution on each client**
Each client will only allow the server to ask it to execute a particular set of tasks, enumerated by the `.job` Inferno shell scripts held in `/grid/slave` on each client. There is a library of shell scripts for possible tasks in the directory `/grid/slave/tasks`, and a given task is enabled on a client by copying the task's prototype `.job` file from there into the parent `/grid/slave` directory. In this example, we are going to allow anything to be executed. The `.job` script that allows this is called `test.job` in `/grid/slave/tasks/test.job`. It is actually quite simple:

```
load std
fn runtask {
        $* > result
}
fn submit {
        cat result
}
```

To enable the task, copy its `.job` file:

```
cp /grid/slave/tasks/test.job /grid/slave
```

4 **Start the grid running**
Start the server software and the Windows client software service on each machine, as detailed in the installation instructions.

2.1. Check that the grid is working

We shall now start a simple job on the grid to check that everything is functioning as it should. A common form of task involves reading some data as records or files, processing it, and writing out the results. The job takes a large set of records (or files), splits that into smaller subsets, one per task, and submits them to the scheduler, which hands them out to suitable workers on request, and later retrieves the results. The final result is a set of result records (or files) each corresponding to the result of one task. No task's output is duplicated in the final result, even if a task had to be restarted or repeated elsewhere because of a failure.

For the first example, we shall have the grid process a file, treating each line of text in the file as a separate task to be executed. The task we run will produce the MD5 checksum of the text on each line, a very quick task, but one that enables us to easily check the correlation between input and output. (MD5 is a mathematical ''checksum'' function that produces a number based on its input data highly unlikely to have been produced from any other data). For this initial test, we are going to work inside the Inferno environment, because that is the same everywhere. Later, we shall show how to run the host's own applications (ie, executables) outside that environment.

1 **Start the Inferno window manager**
On a convenient client machine, double-click on the Vita Nuova icon to start the Inferno window manager.

2 **Start the Grid monitors**
Click on the icon at the bottom-left of the Inferno window, to bring up the application menu, and click on ''Grid'' to bring up another menu with a choice of two monitor programs, one for 'nodes' (ie, connected worker machines), and another for jobs. Select `Node Monitor` first. If everything has worked OK so far, this should connect to the Inferno grid scheduler, and show the currently connected clients. If that works, you can start the `Job Monitor` in a similar way.

3 **Create the job input parameters**
Now bring up the Inferno application menu as before, and click the `Shell` entry to start an Inferno shell window. All that is needed for our example is a simple, small text file to act as input. We could create such a file ourselves, but for a test it is easier just to copy an arbitrary existing file, for example:

```
cp /NOTICE /tmp/mytasks
```

We shall get the grid to process each line in this file, and produce the results in the file `/tmp/mytasks.result`.

4 **Start the job**
There are two ways of doing this, using either the Job Monitor application or through the command-line. Using the Job Monitor, click on the button the`New`at and select `Generic job` from the resulting window. In the panel that appears, enter the following into the text area labelled `Command`:

```
filesplit -l /tmp/mytasks test md5sum
```

You can optionally enter a description such as 'first test job' into the `Description` box. Then click `Start job` to queue the job for the scheduler.

To explain the actual job that we've started: the `filesplit` job type takes a file (its first argument), splits it into multiple records — here the `-l` option tells it that the records are newline separated — and arranges for each record to be given as input to the task that given by subsequent arguments. In this case the `test` task is run, which will invoke the previously-installed script `test.job` on each client, with argument `md5sum`. Hence we have arranged for the `md5sum` command to run on each client. By convention, `filesplit` puts the combined results in a file named after the input file name (ie, `/tmp/mytasks` above) with `.result` appended to it. Thus in the example, the results will be left in `/tmp/mytasks.result`.

Given that we haven't got many tasks, and none will take much time, this job should not take long. The Job Monitor application should show the new job as 'running'. The Job Monitor updates the display periodically itself, but if necessary click on the circular arrows to refresh the display immediately to ensure an up-to-date view. Click on the new job to show its running statistics and progress.

Alternatively, you can start the job from the Inferno command line. From the Inferno shell window, first make the scheduler available in the namepace, and then start the new job, substituting the actual address of the scheduler for *schedaddr*:

```
% run /lib/sh/sched
% mountsched
% start filesplit -l /tmp/mytasks test md5sum
1
%
```

The first two commands are needed only once per session. The file `/lib/sh/sched` contains some Inferno shell functions that simplify interacting with the scheduler from the Inferno shell. One of those functions is `mountsched`, which reads the client configuration, locates the scheduler, and makes it available in the directory hierarchy of that shell window. You can see the scheduler's file name space using:

```
lc /n/remote
```

Another shell function is `start`, which creates a new job with the given parameters (ie, ''`filesplit` etc.''). It prints the resulting job number (1, in the case above).

5 **Check the job output**
You should now be able to see the results of the job in the file `/tmp/mytasks.result`:

```
% cat /tmp/mytasks.result
data 33 2
68b329da9893e34099c7d8ad5cb9c940
data 33 0
af325e1d9e7df186ec43b7064971e591
data 33 3
b1ab8481a01ba86bc936c679f9d09187
data 33 1
bd04ba4c99cb67c7371b6a2c359d7391
[...]
```

Note that the output format is different from the input format, consisting of records in a simple record format, each record with a header line giving the number of bytes of data in the record, followed by the data itself, in this case the output of the md5sum command for each line of input data. Because the tasks are done in parallel, and complete in unpredictable order, the order of the records can be mixed up — the second number in each record header gives the record number of the input corresponding to that output record. So, in the case above we see that task number 2 finished first, followed by task number 0, etc. We can verify that the results correspond to our expectations, by running the md5sum command locally; for instance, we can try the fourth line of the input (record number 3):

```
% echo 'You may copy and redistribute the package as a whole,' | md5sum
b1ab8481a01ba86bc936c679f9d09187
%
```

The grid is now ready to be used. In this configuration, it assumes a homogeneous collection of software available on each node (if one node lacks the software, it will be quickly be blacklisted, as it will be unable to produce the desired results). It is possible to get this grid to execute arbitrary Inferno shell scripts, rather than merely simple commands. Note that the Inferno shell is used by the Inferno grid software in order to express workflow portably across many different operating systems and hardware platforms. For instance, we could extend the example earlier to make the tasks take longer:

```
start filesplit -l /tmp/mytasks test {
        md5sum
        sleep 5
}
```

The output of the job will be the same (execution order aside), but we get each node to sleep for five seconds, so we can (perhaps) more easily see some of the speed-up given by concurrent computation on the grid.

2.2. Running your own job

For the time being, we'll only consider jobs that can be be shoehorned into the above form. Such a job consists of a set of independent tasks, each of which is characterised solely by its input data. If the task simply reads standard input and writes to its standard output, things are easy — all that is needed is to substitute for `md5sum` the name of the command to run. If the command does not run directly under Inferno (the usual case), you can use Inferno's *os*(1) command to run a host-OS program (see below for an example).

For most programs it will be necessary to construct an input data file containing records as described in the earlier discussion on output format. The format is simple enough that it might be easiest to generate this yourself in the scripting language of your choice. Alternatively, there are two commands available on the Inferno command-line, `owen/file2rec` and `owen/rec2file`; the former reads the data in its argument filenames and produces a file of records, each record containing the data from its corresponding file; the latter writes a file for each record that it finds in its standard input.

It is quite common for commands to take files as input in addition to (or instead of) standard input. This can be arranged by constructing a task's input data as a file archive containing all the necessary files. Here is an example where we shall distribute the Windows binary along with the data which it uses to compute. Bear in mind that in this example we are acting in a trusted viroment; if the server is not to be trusted, then clients should definitely not have the `test.job` script installed! Also note that the following technique will not work (without change) where clients run different operating systems or use different binary architectures. We assume for this example that the binary-to-be-executed has been copied to `/tmp/myprog.exe` and that the input files for each task have been placed in the directories `/tmp/task/0`, `/tmp/task/1`, etc. Further, we assume for simplicity that all the command's results are written to its standard output.

First, we prepare a special file that contains the input for all the tasks in the job, including copies of the executable for each task. (In practice, especially if the executable is large, you would access it from a shared file store, or copy it once to each client.) At an Inferno shell prompt, type these commands:

```
% cd /tmp/task
% mkdir record  records
% for(i in [0-9]*){
        mkdir record/data
        cp -r $i/* record/data
        cp /tmp/myprog.exe record
        cd record
        puttar . > /tmp/task/records/$i
        cd ..
        rm -r record/*
}
% cd /tmp/task/records
% owen/file2rec * > /tmp/mytasks
%
```

Now we have a single file (`/tmp/mytasks`) containing all the input data, each record being a *tar* archive containing the executable to run and the input files for that executable (in the `data` directory.

We shall start the job from the Inferno command line, in an Inferno shell window in which the scheduler is available. In a new window, type the following:

```
run /lib/sh/sched
mountsched
```

which makes the scheduler visible as described above. Now start the job:

```
% start filesplit /tmp/mytasks test {
        gettar
        hostwork = $emuroot/$work
        os -n -d $hostwork/data $hostwork/myprog.exe < /dev/null

}
%
```

Before executing each task (the commands between the braces), the shell variable `emuroot` is set to the name of the Inferno directory on that host, and the variable `work` names the temporary directory created to run the task. Putting them together in `hostwork` creates a name that host system commands (such as `myprog.exe` above) can use to refer to directories and files containing the task data inside the Inferno tree. Inferno's `os` command runs a host system command using the host's own command interpreter (shell). It is given two options: `-n`, which runs the command at low priority, and `-d`, which names the host directory that the command should use as its own working (current) directory, `$hostwork/data` in this case, putting it in the same directory as the task's input files. Note that the `os` command has its input redirected from `/dev/null`, when the command `myprog.exe` does not read its standard input. If it did need input, say from a file `myinput` in the task's working directory, we might write:

```
os -n -d $hostwork/data $osroot/myprog.exe <myinput
```

When all the tasks of this job complete, the results are placed in `/tmp/mytasks.result` as before.

3. The `job` task type

There is no particular association of task generator with the actual task performed. For instance, we saw that the task generator `filesplit` reads input records from a file and allows them to be processed by an arbitrary task before writing the results record to another file. In this section, we look at the `job` task type, which uses a compact high-level job description to describe the input data for a specified task, and tells how to collect the resulting output. This is probably the easiest way to use Owen.

TO DO

- complete example using ‘job’
- where do the results go, and how to collect them?
- job submission from command line? perhaps non-Windows only