

Composing workflows in the environmental sciences using Inferno

Jon Blower^{1*}, Keith Haines¹, Adityarajsingh Santokhee¹, Roger Peppé²

¹ Reading e-Science Centre, Environmental Systems Science Centre, Harry Pitt Building, University of Reading, Whiteknights, Reading RG6 6AL

² Vita Nuova Holdings Limited, 3 Innovation Close, York Science Park, York YO10 5ZF

* Corresponding author: email address jdb@mail.nerc-essc.ac.uk

Abstract

In the environmental sciences, there is an ever-increasing need to analyse and visualize very large data sets. Tools based on Grid computing have great potential for assisting the scientist to perform these tasks in an efficient manner. It is a great challenge for the e-Science community to produce suitable tools and applications that are sufficiently easy to use, reliable and responsive. We present here a novel method for creating such a distributed system, with application to a particular case study in oceanography. The main advantages of the system that we have created are: (1) Large data sets can be streamed directly from remote service to remote service, without passing through the client machine or being cached on hard disk; (2) The system seamlessly integrates a pool of computers that are used for processing the data; (3) The system can incorporate existing binary executables, often without modification; (4) The method of specifying the necessary workflows is very simple and intuitive; the user employs a notation very similar to Unix pipes, for example “extract | process | visualize”. The architecture is based around the Inferno operating system.

1 Introduction

Environmental scientists have a need to analyse and visualize large data sets. These data sets commonly come from satellite measurements and numerical analyses and predictions and they contain a huge amount of information about the Earth system and its climate. Data requirements within the environmental sciences are becoming rapidly ever larger. For example, Southampton Oceanography Centre’s OCCAM model of global ocean circulation, which now runs at a spatial resolution of one twelfth of a degree, outputs several hundred gigabytes of data per model year.

These terabyte-scale data sets are typically stored in central repositories and are of interest to a very wide community of climate scientists. There is therefore a need to provide access to these datasets to the community in order to perform the many types of analysis that are required to understand the datasets and the complex systems (e.g. the oceans and atmosphere) that they represent. Visualization of the end re-

sults of such analysis is a very important part of the process [1]. The large amount of data and processing that is required indicates that Grid-like systems based on distributed computing will be of huge benefit to the environmental science community.

These requirements pose some interesting problems for the e-Science community. Among the most important of these are:

- How can we move these large datasets efficiently from computer to computer?
- How can we take advantage of parallel processing where appropriate?
- How can we make systems easy to use?

These problems, naturally, apply to a very wide range of applications outside the environmental sciences as well.

This paper will describe a novel way of constructing and using a distributed system for analysis of environmental data. It is based around the

Inferno operating system, which is a technology that has hitherto not been applied to creating applications in the environmental sciences. As we will show, Inferno has allowed us to create a system that is easy to use, flexible, efficient and powerful.

2 Case study

We have created the system in order to solve a specific problem in oceanography. The root of the problem that we are solving lies in the science of data assimilation.

2.1 Data assimilation

Many modern computer simulations of the atmospheric and oceanic circulation rely on data assimilation to improve their accuracy. This is the process whereby observational data from satellites, ships, buoys, radiosondes etc. are incorporated into a running simulation in order to improve the accuracy of the model [2, 4].

A central problem in data assimilation is that observational data are generally very sparse in space and time and so the maximum amount of information needs to be gained from each observation. Let's say that we have a running computer simulation of global ocean circulation and, at a certain time, we have a point observation of sea surface temperature. At this point and time, the model has a sea surface temperature of 15°C, but the observed temperature is 18°C. Assuming that both the model and the observation have associated known, we might calculate a value of 17°C as the “actual” temperature at this point and continue the simulation from this value.

We can do better than this; since the model has underestimated the true temperature at this point, it is likely that the model's predictions of temperature at other points in the near vicinity are also underestimates. Therefore we would alter the model temperatures of all points within a certain area around the observation. But how big, and what shape, should this area be? Choosing too small an area will effectively throw away vital information, but choosing too large an area will lead to further inaccuracies.

One way of assessing the “area of influence” of an observation at a particular point in space is to see how values of the observation are correlated with values at nearby points over time in the simulation. A high correlation over time between two points implies that an observation at one point can legitimately be used to alter the model's prediction at the other.

A useful exercise, therefore, is to use model data to calculate the correlation of a quantity at a point in space with all the other points in its neighbourhood. These results are then used to plot a one-point correlation map (figure 1(a)).

2.2 Calculation of correlation

The scientist might wish to calculate correlations for many different quantities. Common quantities of interest include the temperature or salinity at a certain depth (T_z or S_z), the depth of an isotherm (z_T), the salinity on an isotherm (S_T) and the temperature or depth of a density surface (T_ρ or z_ρ).

In order to calculate the correlation of a quantity between two points in space, we need to extract the values of this quantity at these points over a period of time. We need to filter out the effects of long-term trends (e.g. global warming) and the seasonal cycle and only calculate correlations based on anomalies.

Having removed the long-term trend and the seasonal cycle, the correlation between the two timeseries is calculated using the formula:

$$\begin{aligned} \text{correlation}(x, y) &= \frac{\text{covariance}(x, y)}{\sigma(x_i)\sigma(y_i)} \\ &= \frac{\langle x_i y_i \rangle - \langle x_i \rangle \langle y_i \rangle}{\sqrt{(\langle x_i^2 \rangle - \langle x_i \rangle^2)(\langle y_i^2 \rangle - \langle y_i \rangle^2)}} \end{aligned} \quad (1)$$

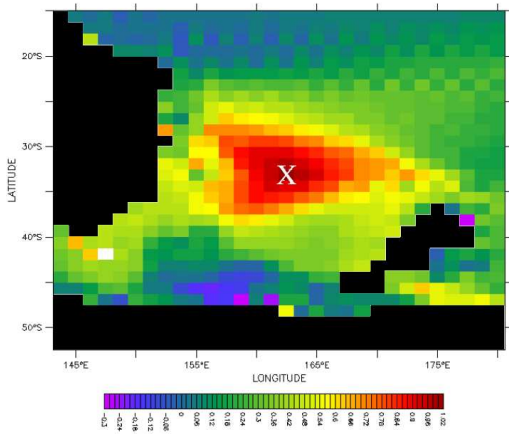
where $\langle x_i \rangle$ is the mean of all the values of the timeseries at point x and $\sigma(x_i)$ is the standard deviation of the values. The value of the correlation lies between -1 and +1.

2.3 Lag correlation

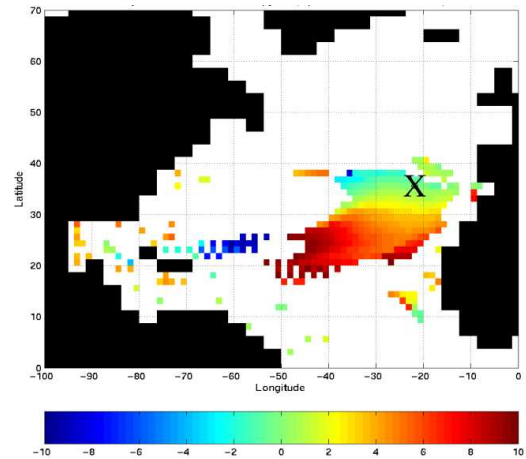
So far we have discussed the calculation of correlation of a certain quantity at two points in space at the same time. The scientist can learn even more about the model by asking questions such as “how does the temperature at point X correlate with the temperature at Y *three months later*?”. If this correlation is high, this implies that an anomaly in temperature at point X is a predictor of variability at Y after three months.

Commonly, a scientist will wish to know the time difference (the *lag*) at which the correlation between two points is at a maximum (figure 1(b)). This reveals a great deal about the dynamics of the system as it allows the scientist to track anomalies through the oceans as circulation progresses.

This adds an extra dimension to the whole calculation. In order to calculate the time of



(a) One-point correlation map showing the correlation of the salinity on the 12°C temperature surface at point X with all other points in its neighbourhood.



(b) Map of the time offset (lag, in years) at which the correlation of the thickness of water between 14°C and 15°C at point X with each point is at a maximum

Figure 1: Different types of correlation map that can be produced by the system

maximum correlation, it will be necessary to calculate the correlation at many different values of the lag, vastly increasing the number of calculations required in the study. However, this particular problem has the great advantage of being trivial to parallelize; each calculation of correlation between two points is totally independent of the others. The execution time of large calculations such as plotting lag correlation maps can be vastly reduced by spreading the load over many computers in a distributed system.

2.4 The dataset

As an initial test, we are using data from the Hadley Centre’s HADCM3 model. At Reading e-Science Centre (ReSC), we currently hold data for 100 model years at 1-month intervals, i.e. a timeseries of 1200 data sets. The data are at a spatial resolution of 1.25 degrees with 20 depth levels. For this study, we require values of both temperature and salinity at all these points. Therefore the total data size is $1200 \times 20 \times 360 \times 180 \times 2/1.25^2 = 2 \times 10^9$ data points, or around 8 GB, given 4 bytes per data point. This is not a huge dataset by any means, but it is large for the types of calculations that we need to perform. In the future we intend to apply the system to much larger data sets at higher spatial resolutions.

3 Constructing the distributed system

In order to perform the kind of investigation described above, one needs to produce one-point and lag correlation maps for many different quantities at any point in space (section 2.2). One also needs to experiment with different methods of interpolation, detrending, temporal filtering and so forth. Any useful system must therefore have the flexibility to allow this experimentation without hindering the scientist. At a high level, the workflow that is required looks like this:

extract timeseries → detrend → temporal filter → calculate correlation → visualize results.

This type of linear workflow of the form “extract data → process → visualize” is very common in the environmental sciences, and no doubt in other fields as well. If we were implementing this on a single machine, we could specify workflows using Unix filters. In this framework, each step in the pipeline is implemented as a separate executable program that reads data from its standard input and writes to its standard output. This would allow the user to write something like “extract | process | visualize”. This gives a method for specifying these linear workflows that is both flexible and familiar to most users, but cannot be directly applied to a distributed system.

We wish the distributed system to be as intuitive and easy to use as if it were implemented on a single machine. As we shall show, we have created such a system that hides its complexities from the user and allows the use of the Unix pipe metaphor for the creation of workflows. We have chosen the Inferno distributed operating system as the base for this application.

3.1 Inferno

Inferno (<http://www.vitanuova.com/inferno>, [3]) is an operating system that is designed from the ground up for distributed computing. Although it can run natively on bare hardware, its lightweight nature means that it can be run as an emulated application on another operating system: versions are available for Windows, Linux, Solaris, FreeBSD, MacOSX and more. In all versions, Inferno presents an absolutely identical environment to its applications. This effectively means that Inferno can be used as a lightweight, robust, platform-neutral middleware layer between machines of different types. When run as an emulated application, Inferno can execute programs in the host operating system. This means that programs can be written in any language and compiled for the host system. They can then be joined together and executed through Inferno. Inferno is free to use (and open source) for applications that are released under the same terms (i.e. also as free software).

A key feature of Inferno is that it represents *everything* as a collection of files. Unix already uses this metaphor to a certain degree: the mouse appears in the operating system as `/dev/mouse` and so forth. Inferno takes this idea to its logical extreme; everything, including physical devices, environment variables and the on-screen display, can be accessed as if it were a group of files. The term for such a logically-related group of “files” is a *namespace*. Inferno uses an open protocol called Styx for all file operations; this is a very lightweight protocol because it only needs to contain operations that pertain to files (open, close, read, write, create, delete, etc).

The beauty of Inferno’s architecture becomes apparent when creating distributed systems. Since everything appears as a file, all that is required to create a distributed system is a method for sharing files across a network. Inferno and its applications use the Styx protocol to interact with files, whether they are local or remote. Therefore, once the system has access to a remote resource (which it does by “mounting” the resource’s namespace, very much like the Unix mount command), it can interact with the

resource exactly as if it were part of the local system. In fact, once the resource is mounted, there is *absolutely* no difference from an application’s point of view between a local and a remote resource (apart, of course, from performance issues that might result from slow network traffic).

The security of the system is assured through Inferno’s built-in mechanisms. Connections between computers can be secured to many different levels. A server can specify that incoming connections have to be authenticated using Inferno’s authentication system, which is based on public key certificates. Furthermore, data connections between computers can be encrypted using a variety of algorithms that include DES, IDEA and RC4 at many different bit depths. Provided that the user has the necessary certificate(s), this authentication and encryption is completely transparent. Once a connection is established to a remote machine (via the mount command), the user or application builder can treat secure and insecure connections in exactly the same way, with Inferno invisibly handling the necessary encryption and decryption of messages.

3.2 Peer-to-peer streaming of data

To recap, we wish to create a system that allows the general workflow “extract | process | visualize” to be executed across a distributed network of machines. Typically, the workflow will be executed by the scientist from a client machine that does not itself do any of the processing. We do not wish the data to pass through the client machine; rather, the system should allow the data to be passed across the network in the most direct way possible.

Furthermore, we would like the data to pass directly from the memory of one machine to the memory of the next machine in the pipeline, without requiring intermediate caching of the data on the hard disk of any machine. In other words, the data should be *streamed* from service to service as it is produced. This allows the services to run concurrently; in our example, the process service can begin processing the first chunk of data that has been extracted, while the extraction of later chunks of data is still in progress. This can make the whole pipeline very much more efficient than if each service had to finish executing before the next one could begin.

Inferno provides a way of specifying pipelines that allows exactly this type of streaming and concurrent execution. It does so through a simple shell environment known as the *alphabet shell*.

3.3 The Alphabet Shell

The alphabet shell is a module that can be loaded into Inferno's default shell environment. It provides many extensions to the shell, the most notable of which, as far as this project is concerned, is the ability to create workflows in which data can be streamed directly from remote service to remote service.

To achieve this streaming, the alphabet shell introduces the concept of an *endpoint*, a place in the network to which a process can go to read a stream of data produced by another process. The process producing the data creates the endpoint, and in doing so gets a unique name (hostname/port/id) for it, sufficient for a remote party to connect to the new endpoint.

This concept can be illustrated through the remote execution server, known as `rexecsrv`. An `rexecsrv` daemon on a machine listens on a specified port number. When an alphabet shell command includes a remote execution step, it connects to `rexecsrv` and passes it an endpoint identifier along with the command to execute. `rexecsrv` creates a new endpoint, returns its identifier to the caller, and runs the command with its standard input and output connected to the respective endpoints.

A client can then filter data through a remote command by creating an endpoint itself, passing its name and the command to `rexecsrv`, and reading back the name of an endpoint from which it can obtain the resulting data. This is done through the client program `rexec`:

```
echo <params> | remote |
rexec tcp!server1!port "{extract}" |
rexec tcp!server2!port "{process}" |
visualize
```

This runs the `extract` and `process` programs on `server1` and `server2` respectively, piping the data directly between the two. (The `remote` module creates an endpoint from which the first remote server can read data.) Behind the scenes, the necessary endpoints are negotiated before the workflow is executed. In the final step, the data are piped back to the local machine for visualization (the local machine provides an endpoint for receiving the data). All machines in the chain are acting as both clients and servers, taking advantage of the inherently peer-to-peer nature of Inferno distributed systems.

Another key feature of the alphabet shell is that it is *strongly typed*. Before allowing a workflow to be executed, it checks that the inputs and outputs of adjacent services are compatible. The

user can define custom types for particular applications, of which `endpoint` is just one example.

3.4 The Inferno Grid

The calculation of correlation over a large area can be a time-consuming task, especially when calculating the correlation with many different lead and lag times. We can take advantage of the inherently parallel nature of the calculation by splitting the region of interest into chunks and processing each chunk on a different machine. There are many tools available for doing this distributed processing (Condor being a very popular example), but Inferno provides a suitable tool of its own.

The Inferno Grid differs from many other similar tools in one important respect. As with most other such systems, there is a *scheduler* that performs the task of supplying jobs to *worker* machines that do the required computations. Whereas most systems use a model of “scheduler-push” to distribute jobs, the Inferno Grid uses a model of “worker-pull”. That is to say, the worker machines *request* work from the scheduler, rather than waiting for the scheduler to send a job to them. This means that the scheduler does not need to continually monitor the state of the worker machines and the workers can decide for themselves when they are ready to accept jobs. It is very easy for workers in diverse domains to take part in the Grid since the workers do not need to have a permanent connection to the scheduler, and no server programs (with associated firewall holes) need to be installed on the workers. It should also be relatively easy to join together (or *federate*) different Inferno Grids, a task that can be rather difficult with other scheduling systems.

An Inferno Grid can be incorporated into a workflow in the alphabet shell very easily. The shell defines a module called `farm`, which allows data to be passed to an Inferno Grid in the pipeline. For example, one could type:

```
extract | remote |
farm tcp!mysched!port "{process}" |
visualize
```

This workflow causes data to be extracted from the store and passed to an Inferno Grid scheduler on the machine called `mysched`. The scheduler will automatically split the data into chunks and distribute the chunks to the worker machines, which will run the program called `process` on each chunk. The scheduler will pass each processed chunk, as soon as it becomes available, to the `visualize` service on the local machine.

The progress of the job(s) running on the Grid can be monitored from anywhere on the Internet using Inferno's monitoring software (figure 2). This program mounts the scheduler's namespace, and thus has access to all the relevant information held by the scheduler. It can then display all this information as if it were running on the scheduler machine itself.

4 Bringing it all together

Having discussed the nature of the problem at hand and the advantages and features of the Inferno system, we can now put all this together to define the system and how it is used. We shall do this by building up a typical use case, step by step.

4.1 Extracting the data

All the data required for this study are stored on a machine which we shall call `datastore`. The `datastore` machine runs Inferno as an emulated application, within which an `rexecsrv` server listens for requests for data and passes them to a program called `extract`. Data requests are constructed by passing a set of parameters to the standard input of `extract` in this way:

```
echo 't_z 15 -20 52' | remote |
rexec tcp!datastore!port "{extract} |
...
```

In this example we are extracting the time-series of temperature at a depth of 15 metres, in a region centred at the point of interest, which lies at -20 degrees longitude, 52 degrees latitude.

4.2 Processing the data

The data are processed in the next stage of the pipeline like this:

```
... |
farm tcp!mysched!port "{detrend | ...
... tempfilter | calccorrelation} |
...
```

The scheduler takes care of the task of splitting the input data, which represent the whole region of interest, into chunks. Every time a worker node makes a request for work, the scheduler streams a chunk of data to the worker, and instructs it to run `detrend` and `tempfilter` to preprocess each chunk of data before calculating the correlations. The output from each worker node will be an array of correlation values, which

are then passed to the visualization stage as soon as they are produced by the worker node in question.

4.3 Visualizing the data

In this case study, the quantity of data that is visualized is much smaller than the quantity of data that is processed. Therefore it is usually appropriate for the visualization to be performed on the client's machine. A simple Java program called `plotdata` has been written for this purpose. This program reads its standard input for processed chunks of data, which it then displays. Since the Inferno Grid cannot guarantee the order in which data chunks will be ready for visualizing, the user sees the chunks appear on the screen in apparently random order. Adding the visualization step to the pipeline is simple:

```
... | plotdata
```

4.4 Simplifying the pipeline

The entire command to extract the data, calculate the correlations and display the results is (omitting any parameters for clarity):

```
echo <params> | remote |
rexec tcp!datastore!port "{extract} |
farm tcp!mysched!port "{detrend | ...
... filter | calccorrelation} |
plotdata
```

In fact, this can be further simplified by pre-defining appropriate modules in the alphabet shell, meaning that a user would simply have to type:

```
extract <params> | process | plotdata
```

This is a considerably simpler expression in which unnecessary complexity is hidden from the user. Recall that the alphabet shell arranges the workflow such that the data are streamed directly from service to service without passing through the user's machine or being cached on hard disk.

The physical architecture of the system is shown in figure 3.

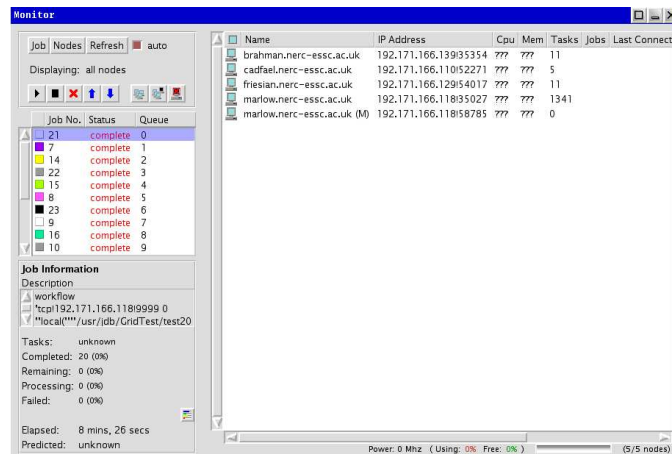


Figure 2: The Inferno Grid monitor. This screenshot shows that, at this particular time, there are four worker nodes participating in the Grid, and one node that is monitoring the Grid (denoted by the '(M)' after the name of the node). The monitor displays information about the progress of jobs currently running on the Grid. Note that in a real system there would be many more worker nodes in the Grid and we are planning to create a much larger Inferno Grid in Reading.

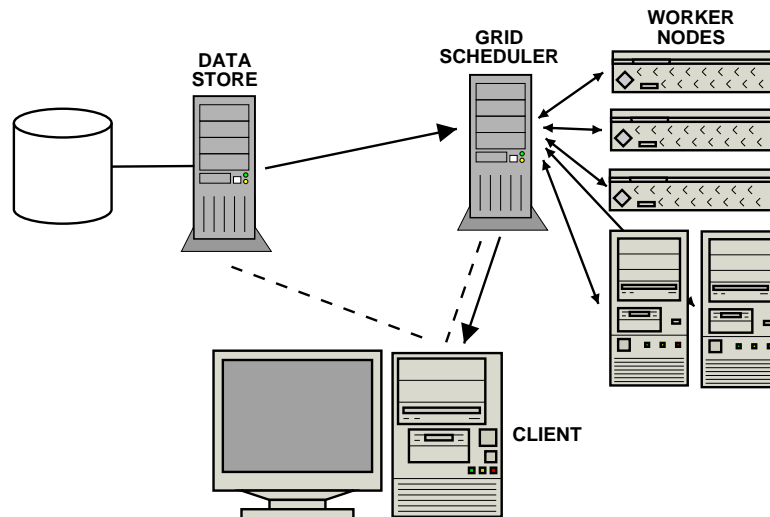


Figure 3: Schematic diagram of the whole system. Dashed lines represent control flow, solid arrowed lines represent data flow. The workflow is composed on the client machine simply by typing an expression such as “extract <params> | process | plotdata” into the alphabet shell (section 3.3). The extract process runs on the machine that holds the data, and the extracted data are streamed directly to the Grid scheduler. Here, the data are split into chunks, which are then processed on a “farm” of worker machines, which may be of many different types. The processed data are streamed directly to the client where a visualization program creates a simple plot of the data.

5 Conclusions and further work

We have created a system, based around the Inferno operating system, that allows time-consuming calculations to be performed across a distributed network of computers in an efficient manner. The key advantages of this system are:

- Data can be streamed directly from remote service to remote service without incurring the overhead of hard disk caching, and without the data passing unnecessarily through the client's machine.
- This direct streaming allows the services to run concurrently; the system can process the first chunks of data while subsequent chunks are still being extracted from the data store.
- Workflows are generated by the user through simple shell commands, which are piped together using a syntax that is very similar to standard Unix pipes. The interface to the system is therefore familiar and intuitive, insulating the user from the complexities of the distributed network.
- The services in the system are created by wrapping existing binary executables. The wrapping itself is largely invisible to the user.

In a Web Services-based architecture, it is more difficult to create such a system with current tools. Typically [1], each service would write its output data to a temporary cache file, and wait for the next downstream service to download this file. This method incurs overhead due to writing the data to the disk, and also loses the benefit of concurrent processing.

In this initial test, the data volumes being processed are not very large by modern standards

of climatological data sets. However, through the use of distributed computing, we have created a system that has the potential to be very much more responsive than one based around a single machine. This allows the user greater freedom to experiment with the system by exploring parameter space, trying new filtering algorithms and so forth. These tasks would be very much more tedious in an environment where computations took tens of minutes, hours or longer to run.

This is very much work in progress, but early experiences are very promising indeed. We could further improve the usability of the system by creating a graphical front end that allows the user to specify workflows using a familiar point-and-click interface.

By wrapping the system in a Python module the system could be incorporated into the popular CDAT system (<http://esg.llnl.gov/cdat/>) for climate data analysis. Users of the system would be able to process their data using the Inferno Grid without necessarily knowing that the Grid was involved.

We are planning to build a much larger Inferno Grid at the University of Reading, and use it to create more applications such as this. We (the ReSC) are evaluating the Inferno Grid and comparing it with other related technologies such as Condor.

This paper was written at an early stage of this project. For updates on progress, a demonstration and more details about the system, please visit the Reading e-Science Centre's website at <http://www.resc.rdg.ac.uk/inferno>.

Acknowledgements

The ReSC would like to thank Vita Nuova Holdings Ltd for working closely with us to ensure that Inferno meets our needs. Much of this work was driven by the NERC e-Science pilot project, GODIVA.

References

- [1] Brooke, J., J.Marsh, S. Pettifer, and L. Sastry: 2004, The importance of locality in the visualization of large data sets. *UK eScience All Hands Meeting, Nottingham University*.
- [2] Daley, R., 1991: *Atmospheric Data Analysis*. Cambridge University Press.
- [3] Dorward, S., R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey, and P. Winterbottom, 1997: The Inferno Operating System. Online: <http://www.vitanuova.com/inferno/papers/bltj.html>.
- [4] Haines, K.: 2003, *Data Assimilation for the Earth System*, Kluwer Academic Publishers, chapter Uses of ocean data assimilation and ocean state estimation. 289–296.