The PGPGrid Project

Paul Cockshott, Lewis Mackenzie, Viktor Yarmolenko

Department of Computing Science, The University of Glasgow, UK

Abstract

A major aim PGPGrid project aims to parallelise the process of extracting range data from an experimental 3D scanner using the Grid as a vehicle for providing necessary resources. The application is potentially highly parallel but has some unusual features such as rapid spawning of processes in real time and a dynamic inter-process network topology. These characteristics are such as to require enhancement of the usual task migration capabilities of the Globus toolkit. The present paper describes firstly, attempts to estimate the real parallelisability of the scanner application and an effort to develop a Java API based on Milner's π-calculus which could be used to extend Globus in the manner required to support systems with a dynamic parallel structure.

1. Introduction

The PGPGrid project has as a core aim parallelisation of the process of extracting range data from the experimental 3D-Matic TV scanner at Glasgow University [1]. It is a joint project with Peppers Ghost Productions, a 3D computer animation company, and the Edinburgh Parallel Computing Centre. The scanner uses 24 synchronised video cameras, each of resolution 640x480 pixels, to image a subject from many directions at once. The data thus acquired is then used to build up a dynamic 3D model with a spatial resolution of around 4mm and a temporal resolution of 0.04seconds. Software has been developed to allow an animator's model to be conformed to data captured from a human actor by the scanner. This provides the opportunity to transfer the detailed movement of a real human subject to the equivalent virtual movement of the model. The conformation software was originally developed to work with still models but the aim of the current work is to extend it to moving models of cartoon-like characters.

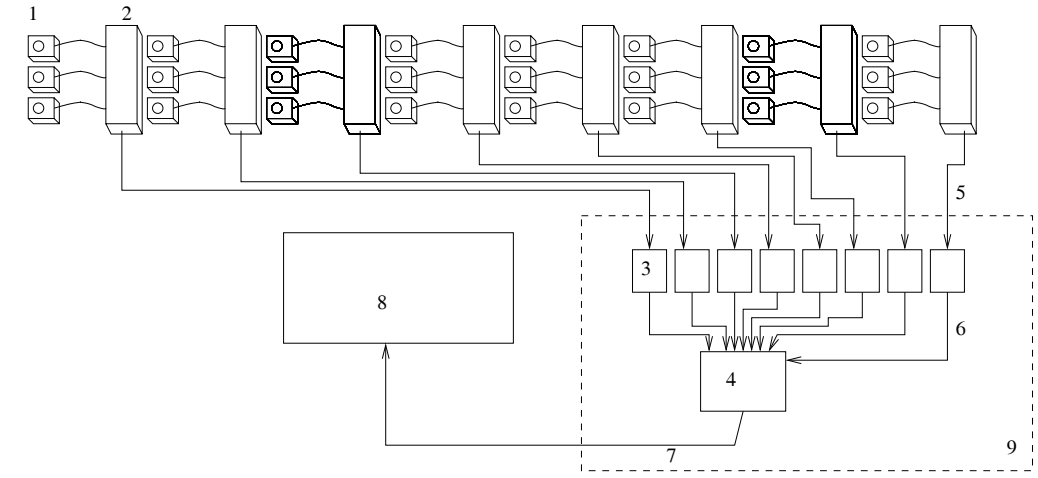
The cameras are organised into groups of three, called *pods,* each consisting of a pair of 8-bit monochrome cameras and a 16-bit single colour camera. All the monochrome cameras are shuttered and synchronised by a 25Hz master trigger signal. A separate 25Hz trigger, with a controllable phase difference from the first, synchronises the colour cameras. The actor is illuminated with speckle pattern from multiple projectors using an uninterrupted light source for the illumination. The speckle pattern provides synthetic features for the computer vision algorithms used in the ranging process to home in on. Without the speckle there may be insufficient contrast in parts of the human anatomy for effective ranging given the equipment currently being used. Strobe lamps synchronised with the colour cameras are set to 'overflash' and thereby suppress the speckle pattern during the exposure of the colour cameras.

Each pod is controlled by a single computer, known as an *Angel*, which captures the three video sequences either to internal RAM buffers, or for longer sequences, to a local RAID disk array. Following capture, each synchronous pair of monochrome images must be processed by a stereo matcher algorithm to produce a *range map* image. Each monochrome image is around 300Kbyes. The colour image is not used for ranging but is required after ranging is complete whereby the images from all 8 pods must be adjusted to blend the brightness of adjacent pixels from distinct views. Each pod therefore generates 1.2Mbytes of data per frame, giving a total data generation rate of over 240Mbytes/second (25 x 8 x 1.2).

The matcher algorithm itself operates via a two stage process as follows

a) A *disparity map* is created for each pod, consisting of three planes, each a 640x480 array of 32-bit values. This amounts to a total of about 3.6Mbytes of data per pod.

b) From each disparity map the range map itself is computed. For each pixel this contains a 32 bit real value specifying the distance in meters from the camera of the corresponding point in the visual field. The map size is thus about 1.2Mbytes in size.



1 POD
2 Capture computer
3 Matcher process
4. Model builder processes
5. Monochrome images
6 Range maps
7. VRML model
8. Control machine
9. Process group

Figure 1: Data-flows and processes for 3D scanner

The range maps from all 8 pods define a “point cloud” of data which is now passed to a model-builder process responsible for creating a triangulated VRML model of the whole subject. Such a model is capable of being rotated and viewed from a variety of directions. Finally the successive VRML models have to be combined into a single space/time model, although this is not an intensive task. Figure 1 summarises the data-flows and processes that are needed to handle each video-frame period of data.

For each frame period (i.e. one capture operation from each pod) it is necessary to initiate a process group comprising 8 matcher tasks and 1 model-building task. In practice, the algorithms used are all implemented in Java. Ideally, a process group of this type should be spawned 25 times a second; however, the matching and model building operations are very CPU-intensive and the processing time required will far exceed the capture time on any feasible system. The Angels each have two AMD

Athlon processors running at 2GHz and it is natural to ask how quickly data could be processed by such systems acting alone. In fact, measurements have shown that a single such processor, with adequate memory resources, can perform a single pod range-map generation in about 50 seconds, with the subsequent building of a VRML model from 8 such maps taking about twice as long as this.

Fortunately, the processes involved are inherently easy to parallelise, although there are substantial challenges involved due to the sheer volume of data being generated and moved between processes. One of the primary goals of this project is to use Grid technology to acquire the resources needed to meet these challenges. The remainder of this paper will discuss our initial attempts to achieve this goal. Sections 2 and 3 look at the degree of parallelisation feasible given the communication resources available. Section 4 discusses a Java interface developed within the context of the project to allow component tasks to be created and dispatched on remote servers and then to form communication channels operating independently of the originating system.

2. Parallelization of the 3D matching algorithm

With regard to parallelisation of the matching and model-building tasks, relationships between the software processes are, of course, crucial. In this regard, two facts are helpful. Firstly, processing from successive frames is inherently independent and secondly, computation of the disparity and range maps for each of the 8 pods for a given frame is also independent. However, the model-building phase for a given frame must be performed on a single CPU and requires substantial data transfer from the matcher processes.

Since the capture process across pods is itself parallel, it seems reasonable for the Angels themselves to initiate the matching operations. They can attempt to run matcher processes of course, thus achieving a certain level of parallelism (8fold), but clearly the CPU-resources available are limited. Assume therefore that a number of process servers are available at various positions on the network and that, via some unspecified mechanism, clients running on the Angels are aware of the locations of all these servers. When a capture is completed by a given pod for a frame, the capture client must locate a free server and submit the data to it (1.2Mbytes) for matching. The server performs the matching and is then in possession of a range map (also 1.2Mbytes) which must be passed to a VRML model building task. This latter is not parallelisable for a single frame (without significant changes to the algorithm), *i*, but since all frames are independent, if one server takes on the task for frame *i*, processing for $i+1$ can begin immediately. If all matcher servers are also able to model-build an extremely efficient asynchronous parallel system can be established. A possible pattern of data exchange between parallel processes and hosts is shown in Figure 2.

At this point two questions arise. The first concerns the limits of such a parallelisation mechanism and here, it is clear that the determining factor will be the speed at which data can be transmitted between servers and clients and between servers and other servers. Some initial experiments have been carried out over local Ethernet and the Glasgow-Edinburgh SuperJanet connection and these are described in Section 3.

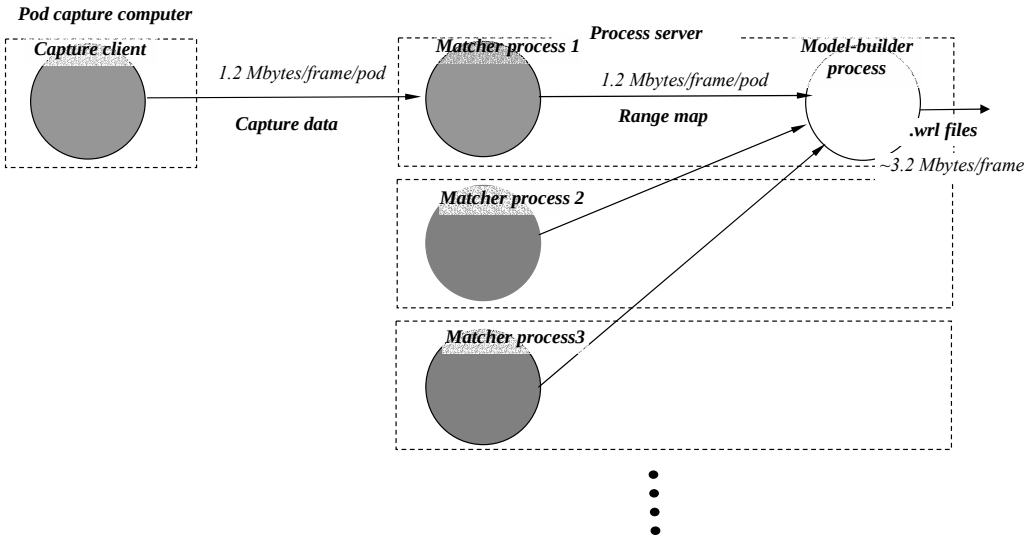


Figure 2: Basic Parallelisation data flows for process servers

The second question concerns the mechanism whereby parallelism can be invoked dynamically across a resource rich network such as the Grid. The frame sequencing generates new matching and model building work continuously. The most effective way to handle such a stream of work in an environment where processing resources are constantly entering and being removed from the available pool is to provide a mechanism for dynamically spawning matching and building tasks remotely on appropriate process servers. Furthermore, some spawned tasks may need to create and maintain data connections to others on other hosts (e.g. matcher servers dispatch the range maps for a frame to the server which has been selected to perform the model build for that frame). These issues can be summarised as respectively requiring an ability to spawn tasks dynamically on any suitable hosts and an ability to transmit data pipes (as connections between processes) over the network with endpoints on different machines. The existing batch-oriented task submission capabilities of the Globus toolkit require some augmentation in order to allow these requirements to be satisfied. A suitable extension, based on Milner's Pi calculus [2] is discussed in Section 4.

3. Preliminary Experiments on Parallelisation

The parallelisation experiments conducted so far have used a reduced version of the process architecture discussed above with relatively pessimistic data flows, statically located matcher processes and a simple server acquisition system. Only four pods and their associated control computers (Angels) have been used (this arrangement is commonly employed for imaging a subject's head only) but instead of capturing data directly from the cameras it is supplied in the form of a pre-recorded 300 frame sequence from local hard disk at the natural rate of one frame every 0.04 seconds . Each capture client, hosted on an Angel, runs a number of job management threads and, as soon as a new pair of monochrome images is acquired, control of processing is passed to one of these. The newly active thread will immediately

attempt to acquire an available matcher server process, although, here, the implementation only proceeds as far as computation of the disparity map from the image pair.

For the purposes of these experiments, matcher processes are already running on each available server host and each client holds a simple static list of the IP numbers and ports for these. A matcher will only accept a connection from a client if it is not already busy; however, once a connection is accepted, input data in the form of 2 monochrome images (~ 600K) is streamed from the client (note that, for simplicity, the colour image is not transmitted). Once the matching process has completed, the whole disparity map (~3.6 Mbytes) is returned to the client. This simplified sequence actually involves the return of significantly more data than would be required were either the range map or model files transmitted instead (the matcher clients here do not perform these steps). After completion of the map transfer, the matcher returns to its original state, i.e. waiting to accept a connection from a client.

This arrangement ensures that all the server hosts are equally utilised and each is running no more that one server task at a time. All work allocation is left to the client so as to minimise the administrative load placed on the servers viewed here as processing workhorses.

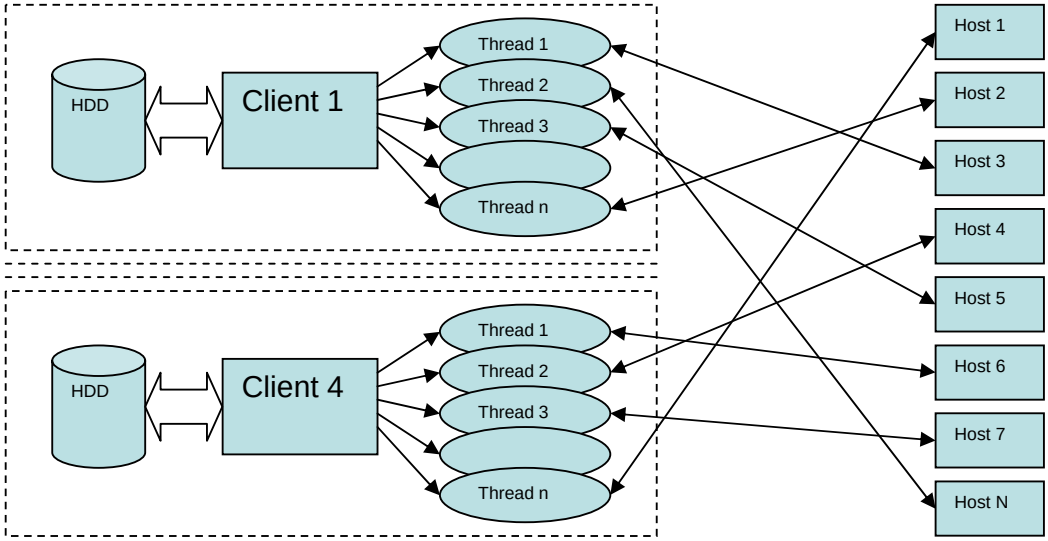


Figure 3 Schematic view of the prototype. Number of threads on each client follow the relation N<4*n

The 4 Angels used and the 4 unused are all dual processor 2GHz Athlon machines. They are attached to a switched 100Mbps switched Ethernet LAN which in turn is linked to SuperJanet via a 1Gbps pipe (see Figure 4).

The server hosts available for running matcher processes fall into three categories

- The Angel machines themselves (being dual-processor, even those running capture clients have spare processing capacity).
- Various other hosts attached directly to the local switched Ethernet.
- A remote IBM 16 CPU host, *Blue Dwarf*, sited at NESC in Edinburgh.

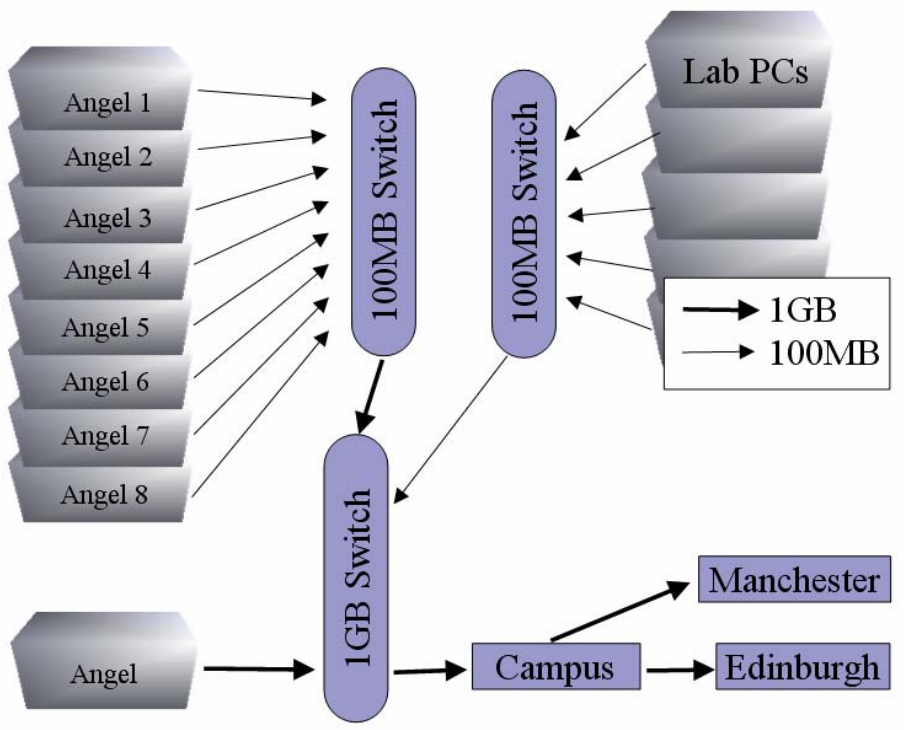


Figure 4: Topology of the Network

Exp	Total CPUs	Angel CPUs	Remote CPUs	Time (min)	Comments
0	4	4	0	215	One Angel CPU/pod
1	12	12	0	75	12 Angel CPUs
2	16	0	16	48	16 NESC blue dwarf CPUs
3	28	12	16	29	16 blue dwarf plus 12 Angel CPUs
4	43	12	31	21	16 blue dwarf plus 12 Angel plus 15 other servers

Table 1: Results of parallelisation tests

Prior to running parallelisation experiments, the Angel and Blue Dwarf processors were benchmarked. The respective times taken to perform a single disparity map computation from a pair of monochrome images were about 43 and 35 seconds. The first entry in Table 1 (Exp 0) follows immediately from this. Here the four disparity maps for each frame are computed. Since there are 300 frames and there is no network communication involved (the processors used are the second CPUs on the Angel machines themselves) the time taken is 300 x 43 seconds or 215 minutes.

In other cases, however, the network is a critical factor and ultimately limits the amount of parallelism possible. Each Angel is connected to a 100Mbps full

duplex switch and needs to receive 3.6Mbytes returned for each remote task initiated. Assuming the machine can service its network interface at maximum speed and that there is no other network traffic, the best possible time in which an Angel can service a matcher task is about 0.3 seconds. If each matcher task takes n seconds, the total number of servers that an Angel can keep busy is about $3.5n$. This would suggest that for 2GHz class processors running our Java algorithms, parallelisation of about N=150 should be achievable in the tests.

Unfortunately at present there are insufficient potential server hosts available to test this limit (a total of 43 are currently assignable). Nonetheless, there are many assumptions implicit in the estimation process of the last paragraph and it is reasonable to see if the prediction holds at least out to values of N which can be tested in our environment. The results of the parallelisation tests are listed in Table 1. Each row gives the test number, total number of process servers used, the number hosted on the Angel machines themselves, the number hosted remotely and the total time to process the 300 frames.

Given the estimate of Exp 0, Exp 1 simply triples the number of servers and thereby reduces the time taken by just under a factor of 3 indicating an approximately ideal speedup. The small loss (75 minutes rather than 72 minutes) is assumed to be caused to the overheads due to server location and real network transmission (not included in the simple estimate of Exp 0). The 16 Blue Dwarf processors of Exp 3 are individually faster at matching by a factor of about 1.23 times and there are 1.33 times as many so the result expected with an ideal speedup would be 75/(1.33 x 1.23) or 46 minutes. Again this is very much in line with the result obtained. Exp 3 is similarly almost exactly in line with what might be expected.

The final test used all CPU resources which could currently be marshalled to support the project, involving the use of another 15 machines on the local Ethernet of approximately 2GHz capability. An ideal speedup calculation on these assumptions yields an expected time of just over 20 minutes, again very close to that actually observed.

Within the range of the CPU resources available therefore it seems clear that the frame processing task is indeed parallelisable and a speedup of an order of magnitude is seen for a similar increase in the number of processors. It should be reiterated however that the conditions of the experiment are pessimistic. If matcher servers return range maps instead of disparity maps the amount of data is reduced to 1.2Mbytes per task, effectively tripling the estimated parallelisation limit. This can be improved still further by performing the model-building remotely. With gigabit interfaces a further order of magnitude increase is theoretically achievable. Overall, parallelisation of the order of 10^3 would seem feasible with a sufficiently high bandwidth network. Such a level of CPU power, while currently not available, would make real time processing of 3D images a realistic possible.

4. The JPie Interface

As discussed at the end of Section 2, the dynamic task creation and remote pipe migration required by this application has motivated the development of a Java interface loosely modelled on the primitives of the π calculus [2, 3]. This interface, *JPie*, is intended for use as a substratum for GRID-based parallel computing applications of this type. It must allow the dynamic creation of remote processes and communications channels between such processes. It should also allow dynamic

reconfiguration of the network of such channels existing between processes. JPie aims to achieve this with the minimum number of primitives and to integrate these into the existing Java class framework. It is at roughly the same abstraction level as JPVM [4] and IceT [5].

The overall JPie class hierarchy is as follows

1. `abstract class JPieTask` represents the unit of work to be done a parallel process. It implements the `java.lang.Runnable` interface, allowing the construction of threads from `JPieTasks`. It has additional methods that allow channels to be associated with `JPieTasks`. Before the `JPieTask run` method is called, the task's transmissible input and output channels, known respectively as *funnels* and *taps*, must be initialised. A running task can be obtain its taps and funnels via `get` methods supplied.

2. `interface JPie` is a factory interface whose job is to create processes, taps, funnels and pipes (linked tap and funnel). It has a core method `spawn` which causes a `JPieTask` to be run either locally or remotely depending on resources. Time and memory parameters which specify the anticipated resource usage of the task are supplied. `spawn` will fail if adequate resources cannot be found.

   JPie also includes methods to create taps and funnels that can convert streams in the local environment to JPie streams that can be sent to remote machines. For example, a local spawning task can create a pipe, pass the input end (funnel) to a remote process and then use output end (tap) to write data to that process. The reverse configuration allows the spawning task to get results back from the remote task. The `createPipe` method also allows interprocess communication streams to be set up between two daughter tasks.

   The JPie interface is implemented in two ways: by `class UniprocessorJPie` which uses local resources and runs the tasks in local threads; and by `class GridJPie` which will use Grid protocols to transfer tasks to remote machines to be run. In the latter case, the set of classes required by a remote task are made available to it via the URL of a set of jar files pre-loaded onto a web server prior to running the JPie application. Except for the standard JRE and JPie classes only classes in these jar files are permitted in the application class path. This is to ensure the unambiguous resolution of class names across the distributed application.

   There is also a need to be able to identify sources of data that are at known locations on the Web. We provide for this by adding methods that allow for taps and funnels to be published and associated with URIs. Each machine running `GridJPie` has a servelet thread running GRID FTP protocols [6]. This thread can respond to requests to get data from published taps or put data into published funnels. When a tap or funnel is published the public name is combined with the machine's internet name or IP address to form a virtual public file name.

3. `abstract class JPieTap` implements `java.io.InputStream`, and requires the implementation of a method to get a byte (read). The standard Java blocking protocols are followed.

4. `abstract class JPieFunnel` implements `java.io.OutputStream` and must provide a write method that writes a single byte to the output channel. It can block on write if the remote process has not performed sufficient reads.

5. `interface JPiePipe` provides a bi-directional inter task communications channel.

This has two methods: `getTap()` and `getFunnel()`. To set up a data flow graph one creates pipes and then passes the input and output channels to tasks which are then forked. It is implemented by class `UniprocessorJPiePipe` using the built in Java classes `PipedInputStream` and `PipedOutputStream`; and by class `GridJPiePipe` using an appropriate remote communications protocol. One of the key issues here is to ensure that pipes, taps and funnels can be serialised.

The overall aim of JPie is for processes to be able to run across a network of machines without knowing where these machines are or having to reference the machines explicitly in any algorithm. An algorithm expressed in Java using JPie should be invariant under alterations in the physical collection of machines that run it.

In principle one of the great strengths of Java is that it allows a single binary image to run on a variety of different physical computers, each of which may be running a different operating system and using a distinct instruction set. In order to ensure that a task will produce the same result regardless of its host, every machine that is to run JPie tasks must have the JRE installed and have access to the appropriate .jar files for the application containing the code to be run. These jar files are mounted on a web server making them generally accessible to any java VM with an internet connection. Other data required by *T* can be passed as `Tap` and `Funnel` parameters to *T* which can then read its inputs from taps and write its results to funnels.

Consider the problem of creating on local machine, *A*, a task, *T*, that will execute remotely, filter a source file stored on *A* and write its result to a destination file on *A*. When *T* is spawned the initiating task on *A* first locates a machine *B* able to run it. On *B* a daemon is executing which accepts task execution requests. Once the daemon agrees to run the job, *A* sends it the URLs of the jar files that contain the classes used by *T*. The daemon execs a new java VM with the class path set to include these URLs and provides, on the command line, details of the socket *S* from which the task is to be read.

The machine A then serialises *T*, through *S*. The new VM on B runs a task launcher class which deserialises the task object from *S*, and runs it. Since the originating VM on *A* and the new VM on *B* share the same class path, the task object on socket S can be successfully deserialised. The task object however contains Taps and Funnels, which must themselves be serialised. The serialization of JPieFunnel and JPieTap classes is achieved by `linking' the serialised object to the original stream by various means that depend on implementation (File, VM memory, TCP, GRID, etc) and details of which we omit in this paper. We only mention that from the perspective of JPie interface, there is no difference.

Note, the JPiePipe can be serialized conventionally, since all of the fields it contains are serializable. This means that if an instance of JPiePipe is created on the original VM and then passed to two separate tasks on remote VMs and they use each of the ends of JPiePipe to stream data, then the data will travel via the original VM. Alternatively, if a remote task creates an instance of JPiePipe and passes it to the original VM, which in turn passes it to another remote task (where the first and the second remote tasks use each end of this JPiePipe) then the data will be streamed directly from the first remote task one to the second, bypassing the original VM. In other words, no matter how many times JPieFunnel or JPieTap are passed (either separately or as a part of JPiePipe), upon their deserialization they will always be reconnected with VM of their origin.

To date the JPie interface has been implemented in uniprocessor form and in multiprocessor form using TCP sockets as the communications API. A fully Grid-compliant version is under development.

Conclusions

The parallelisation experiments described above show that the 3D scanner application is extremely well-suited for distribution in a Grid-type environment. With sufficient resources, parallelisation of sufficient degree to support real-time processing of images can reasonably be envisaged. However, the dynamic nature of the process creation and dispatching requirements and the necessity of supporting a reconfigurable inter-process communications network requires an extension to the normal Globus task control capabilities. With this in mind the authors have described a Java interface modelled on the pi-calculus which satisfies these requirements. This has been successfully implemented in a preliminary form and work is proceeding to integrate it fully with the Grid protocols.

Acknowledgments

The authors wish to acknowledge the funding by NeSc, Edinburgh, UK, and thank our project partners Peppers Ghost Productions and The Edinburgh Parallel Computing Centre for their input.

Bibliography

1. Experimental 3D TV studio, Cockshott, W.P., Hoff, S., Nebel, J-C, IEE Proceedings - Vision Image and Signal Processing, 2003.
2. "The Polyadic π-calculus, a tutorial", Milner, R., 1991.
3. “A calculus of mobile processes”, Milner, R., Parrow, J., Walker, D., Information and Computation, 100:1-77, 1992.
4. JPVM: Network Parallel Computing in Java, Ferrari, A., ACM 1998 Workshop on Java for High Performance Network Computing.
5. IceT Distributed Computing and Java, Gray, P.,Sunderam, V., Concurrency, Practice and Experience, Vol. 9, No. 11, Nov. 1997.
6. GRID FTP Universal Data Transfer for the GRID, Globus Project White Paper, Unversity of Chicago, 2000.