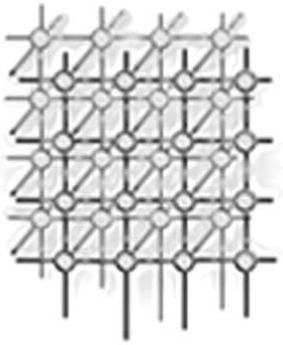
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **19**:235–249
Published online 11 May 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1043



Applying the Grid to 3D capture technology

Lewis Mackenzie[1], Paul Cockshott[1,*,†], Viktor Yarmolenko[1], Ewan Borland[1], Paul Graham[2] and Kostas Kavoussanakis[2]

[1]*Department of Computing Science, 17 Lilybank Gardens, University of Glasgow, Glasgow G12 8QQ, U.K.*
[2]*Edinburgh Parallel Computing Center, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, U.K.*

SUMMARY

The PGPGrid project aims to parallelize the process of extracting range data from an experimental 3D scanner using the Grid as a vehicle for accessing 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 initially discusses attempts to estimate the real parallelizability of the scanner application. It then describes a new Java application programming interface, based on Milner's π-calculus, which could be used to extend Globus in a manner capable of supporting systems with this kind of dynamic parallel structure. The location of processing resources for the π-calculus is done using a Web-services-based resource locator. The article also describes the pipeline of processing from initial stereo photogrametry to the final production of animation models. A key step in this is the conformation of animators models to the data obtained by the real-time scanner. Algorithmic innovations in this process are described. Copyright © 2006 John Wiley & Sons, Ltd.

Received 10 January 2005; Revised 8 June 2005; Accepted 24 August 2005

KEY WORDS: Grid; π-calculus; Java; animation; 3D capture

1. INTRODUCTION

PGPGrid, a joint venture with Peppers Ghost Productions and the Edinburgh Parallel Computing Centre (EPCC), aims to parallelize extraction of range data from Glasgow University's 3D-Matic TV scanner [1], which uses 24 video cameras to image a subject from many directions at once.

*Correspondence to: Paul Cockshott, Department of Computer Science, 17 Lilybank Garden, University of Glasgow, Glasgow G12 8QQ, U.K.
†E-mail: wpc@dcs.gle.ac.uk

Contract/grant sponsor: NESC

The data are then used to build a dynamic 3D model with a spatial resolution of 4 mm and temporal resolution of 0.04 s. Software allows an animator's model to be conformed to data captured from an actor, transferring the movement of a real human subject to equivalent virtual movement of the model. The conformation software was developed for 'still' models but current work extends it to moving cartoon-like characters.

The stereo-pair images collected by the cameras are processed using photogrammetric techniques developed by the Turing Institute, Glasgow, to create a spatio-temporal 3D model of this space. The concept was based on the culmination of over a decade of research into 3D imaging by the Turing Institute and subsequently by the 3D-MATIC Faraday Partnership (operating within the Computing Science Department at Glasgow University). The Turing Institute has developed 3D imaging software [2,3], called C3D, which processes stereo-pair images to generate 3D models. In fact, C3D automates the processing of several stereo-pair views of a subject into a set of 3D views, which are then integrated into a single model using the marching cubes algorithm [4].

Equipment previously developed by the Turing Institute had included an upper-torso scanner developed for Ealing Studios, which used a collection of 15 digital cameras to build models of the upper bodies of subjects. The basic component of this earlier system was termed a pod and comprised a group of three digital cameras, two of which were monochrome and one of which was colour. The monochrome cameras were used to extract range data and the colour camera was used to record skin, clothes and hair texture. The algorithm attempts to find the disparity between the positions of a given real world object as viewed from two cameras. This earlier work used still cameras.

The cameras in the current system are still organized into *pods*, each comprising two 8-bit monochrome cameras and a single 16-bit colour camera. The monochrome cameras are synchronized by a 25 Hz master trigger signal. A separate trigger, with a phase lagger, synchronizes the colour cameras. Using continuous light, the actor is illuminated with a speckle pattern, which provides features for the ranging algorithm to home in on. Without speckle there is insufficient contrast for effective ranging. Strobe lamps synchronized with the colour cameras overflash suppressing the speckle during the exposure of the colour cameras.

Each pod is controlled by a single computer that captures the three video sequences to internal RAM buffers. Next, each synchronous pair of monochrome images is matched to produce a *range map*. Each monochrome image is around 300 KB. The colour image is used afterwards, when images are adjusted to blend the brightness of adjacent pixels from distinct views. Each pod generates 1.2 MB of data per frame, a total data-generation rate of over 240 MB s^{-1}.

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

(a) A *disparity map* is created for each pod, consisting of three floating point planes, a total of about 3.6 MB per pod.
(b) From each disparity map a range map is computed. For each pixel this contains a 32-bit float giving the distance in meters from the camera. The map is about 1.2 MB in size.

The range maps from eight pods give point cloud data to a model builder to create a triangulated virtual reality modelling language (VRML) model. Finally successive VRML models have to be combined into a single space/time model (Figure 1). For each frame period we initiate a process group comprising eight matcher tasks and one model-building task. The algorithms are all implemented in Java. Ideally, a process group of this type should be spawned at 25 Hz; however, the operations are very CPU-intensive and the processing time exceeds the capture time on any feasible system.

DOI: 10.1002/cpe

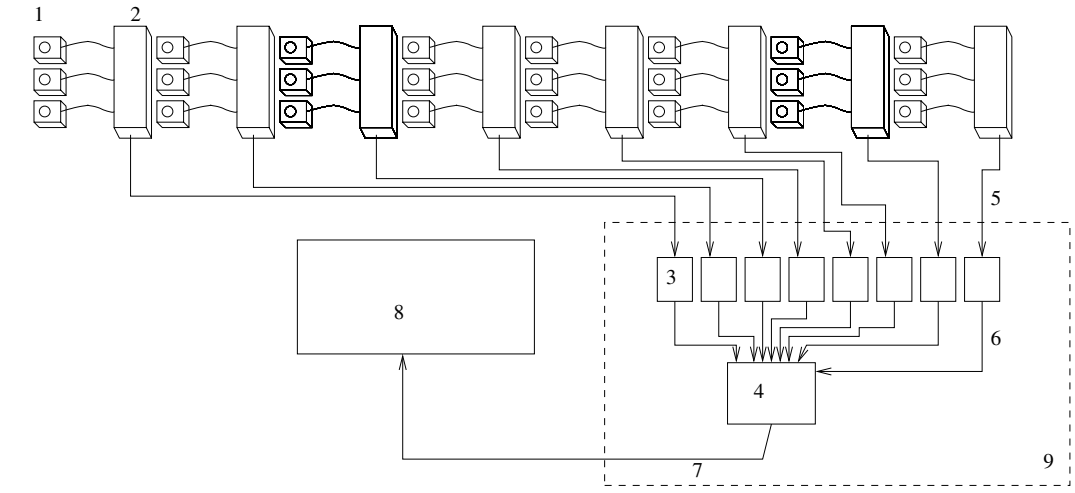


Figure 1. Data-flows and processes for a 3D scanner.

The capture PCs have two 2 GHz Athlons, a single processor of this class, with adequate memory resources, can process one pod range map in 50 s, the VRML model build from eight such maps taking 100 s.

The processes are inherently easy to parallelize, the main challenges being the sheer volume of data being generated and moved. One of our goals is to use Grid technology to acquire the resources needed. The remainder of this paper will discuss our initial attempts to achieve this goal.

2. PRELIMINARY EXPERIMENTS ON PARALLELIZATION

The parallelization experiments conducted so far have used the process architecture in Figure 2 with statically located matcher processes and a simple server acquisition system. Four pods and their associated control computers are used, imaging a subject above the shoulders only. Instead of capturing data directly from the cameras, a pre-recorded 300-frame sequence is streamed from the local hard disk. Each capture client runs a number of job management threads and, as soon as a new pair of monochrome images is read, control of processing is passed to one of these. The newly active thread immediately attempts to acquire an available matcher server process.

Matcher processes are always running on each available server host and each client holds a static list of the IP numbers and ports. A matcher will only accept a connection from a client when not already busy. Once a connection is accepted, input data (∼600 KB) is streamed from the client. The colour image is not transmitted. When the matching process has completed, the disparity map (∼3.6 MB) is

DOI: 10.1002/cpe

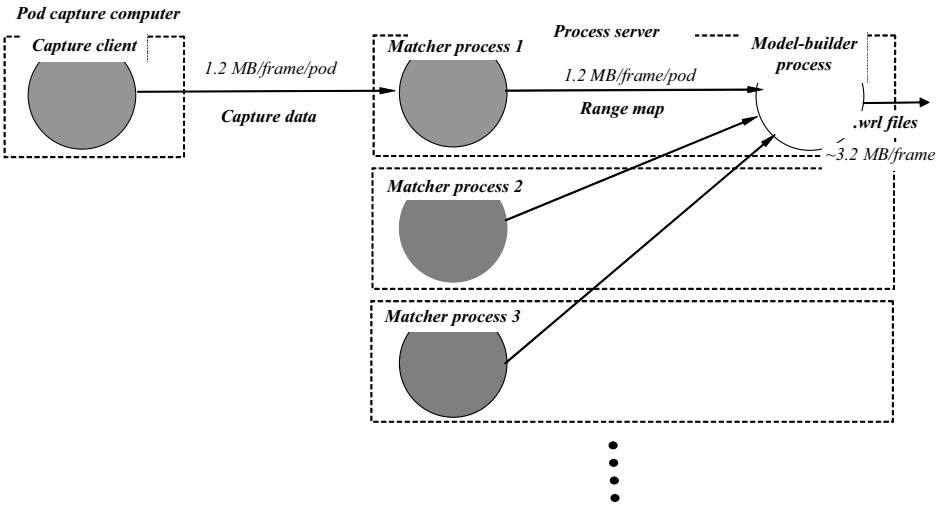


Figure 2. Basic parallelization data flows for the process server.

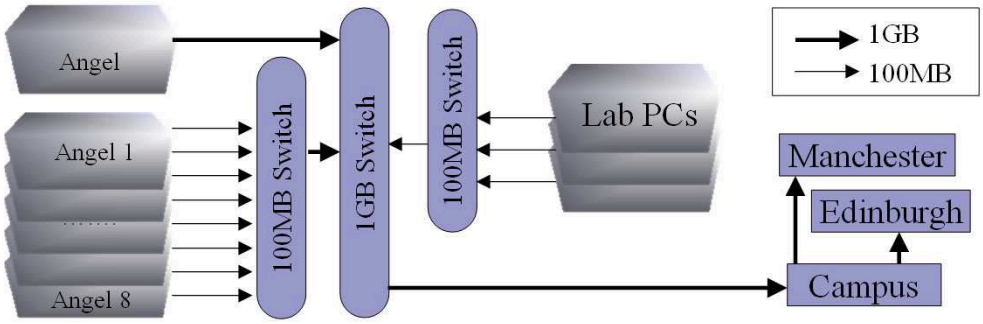


Figure 3. Topology of the network.

returned to the client, and the matcher waits for a new connection. This arrangement ensures that all the server hosts are equally utilized and each runs no more that one server task at a time. All work allocation is left to the client to minimize the load on the servers. The capture machines are attached to a 100 Mbps Ethernet LAN linked to SuperJanet via a 1 Gbps pipe (see Figure 3).

The server hosts available for running matcher processes fall into the following categories: the eight capture machines (even those running capture clients have spare capacity); various hosts attached directly to the local switched Ethernet; and a remote IBM 16 CPU host, Blue Dwarf, sited at NESC in Edinburgh.

DOI: 10.1002/cpe

Table I. Result of parallelization tests.

Exp	Total CPUs	Capture CPUs	Remote CPUs	Time (min)	Normalized CPU resource	Parallel efficiency	Comments
0	4	4	0	215	1	100% (base)	One Angel CPU per pod
1	12	12	0	75	3	96%	12 Angel CPUs
2	16	0	16	48	4.9	91%	16 NESC Blue Dwarf CPUs
3	28	12	16	29	7.9	97%	16 Blue Dwarf plus 12 Angel CPUs
4	43	12	31	21	∼10.7	95%	16 Blue Dwarf plus 12 Angel plus 15 other servers

Prior to running parallelization 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 s (first entry in Table I, Experiment 0).

At this point two questions arise. The first concerns the limits of such a parallelization 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.

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 that has been selected to perform the model build for that frame). These issues can be summarized 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 π-calculus: is discussed in Section 4.

The network is a critical factor and ultimately limits the amount of parallelism possible. Each Angel is connected to a 100 Mbps full duplex switch and needs to receive 3.6 MB 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 dispatch a matcher task is about 0.3 s. 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 the 2 GHz class processors running our Java algorithms, parallelization of about $N = 150$ should be achievable.

At present there are insufficient server hosts to test this limit (a total of 43 are currently assignable). Nonetheless, it is interesting to see if the prediction holds at least out to values of N which can be tested. The results of the parallelization experiments with various groups of processors are listed in Table I.

DOI: 10.1002/cpe

The Parallel efficiency entry for a given experiment is the actual speedup divided by the best speedup calculated on the basis solely of the CPU resources allocated, taking Experiment 0 (with four Angel processors) as baseline. The normalized CPU resource entry compares the CPU resource available in an experiment with that used in Experiment 0, taking into to account differences in performance of the different CPU types used when benchmarked for our application.

As can be seen, with the CPU resources available, the frame-processing task is parallelizable and a speedup of an order of magnitude is demonstrated for a similar increase in the number of processors. The conditions of the experiment are pessimistic. Were matcher servers to return range maps instead of disparity maps, data would be reduced to 1.2 MB per task, tripling the parallelization limit. This can be improved further by also building the models remotely. With gigabit interfaces, parallelization of the order of 10^3 seems possible. Such levels of processing power and backbone bandwidth are not yet realistic, but the argument suggests that real time processing of 3D images will be possible.

3. THE JPIE INTERFACE

The dynamic task creation and remote pipe migration required by PGPGrid motivates the development of a Java interface loosely modelled on the primitives of the π-calculus [5,6]. The interface, JPie, is intended for use as a substratum for this and other Grid-based parallel computing applications. It allows the dynamic creation of both remote processes and communications channels between them. It also allows dynamic reconfiguration of the network of channels. JPie tries to achieve this with the minimum number of primitives, integrating these into the existing Java class framework. It is at roughly the same abstraction level as JPVM [7], IceT [8] and JCSP [9].

The overall JPie class hierarchy is as follows.

(1) Abstract class JPieTask represents the unit of work to be done as 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 initialized. A running task can obtain its taps and funnels via the get methods supplied.

(2) Interface JPie is a factory interface whose job is to create processes, taps, funnels and pipes (a pipe is a linked tap and funnel). It has a core method spawn, which causes a JPieTask to be run locally or remotely depending on resources. Time and memory parameters that specify: anticipated resource usage 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. The createPipe method also permits interprocess communication streams to be set up between two daughter tasks.
The set of classes required by a remote task are made available via a custom classloader which retrieves jar-files containing class definitions from the originating process. The set of jar-files available to the classloader is specified via a call to the setJars(. . .) method of the originating JPie instance.
There is also a need to be able to identify sources of data that are at known locations on the Web. JPie will provide for this by adding methods that allow for taps and funnels to be published and associated with URIs.

DOI: 10.1002/cpe

(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 passes the input and output channels to tasks which are then forked. It is implemented by class *UniprocessorJPiePipe* using built-in Java classes *PipedInputStream* and *PipedOutputStream*, and used by class *GridJPiePipe* using an appropriate remote communications protocol. A key issue here is serialization of pipes, taps and funnels.
(6) 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 JPie library. Data can be transferred to or from a task over its *taps* and *funnels*.

Consider the problem of creating a task T on local machine A 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 that accepts task execution requests. Once the daemon agrees to run the job it executes a new Java virtual machine (VM) and provides, on the command line, details that are used to connect A and B with a pipe P.

The machine A then serializes T, through P. The new VM on B deserializes T using a custom classloader, which loads required class definitions from A, and runs it. T contains Taps and Funnels, which must themselves be serialized. The serialization of JPieFunnel and JPieTap classes is achieved by 'linking' the deserialized object to the original stream. The method used to perform this 'linking' is irrelevant from the perspective of the JPie interface.

Note, the JPiePipe can be serialized conventionally, as all of the fields it contains are serializable. Suppose an instance of JPiePipe is created on an original VM and then passed to two separate tasks on remote VMs. If these tasks use each of the ends of a JPiePipe to stream data, then the data will travel via the original VM. However, 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, 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 deserialization they will always be 'linked' directly to their original VM.

3.1. Resource locator

JPie can make use of various resources from a pool, but requires a mechanism to identify, check for availability and access these resources. This is where the idea of a resource locator comes in.

DOI: 10.1002/cpe

The model we have proposed has three components: the Initiator, the Locator and the Server Entity (ServEnt), jointly referred to as the ILS model. The EPCC is undertaking the major share of the development of this software with requirements and other inputs provided by 3D-Matic Lab.

Figure 4 shows the deployment diagram for the ILS model. The three components are discussed in more detail below, but essentially a typical interaction would be as follows. A ServEnt advertises its hardware characteristics and availability for JPie tasks by informing the Locator, where the Locator maintains a list of such information. An Initiator wishing to launch a JPie task requests suitable resources from the Locator. The Locator returns a list of resources, from which the Initiator then confirms the first resource availability directly with the ServEnt, and then if available submits the JPie task. On task completion the ServEnt passes the job metrics back to the Initiator. Currently the EPCC is focusing on the Locator-ServEnt and Locator-Initiator interactions and functionality.

3.1.1. Locator

The Locator maintains an updateable list of resources with their associated static and dynamic information. Static information includes hardware items such as memory, number of processors and so on, whilst the dynamic information consists of the availability of the resource to accept JPie tasks. The Locator stores this information in a database so that it can be maintained, accessed and searched quickly and efficiently. The Initiator and ServEnt can request and update this information via two Web services [10] made available by the Locator. These are as follows.

- *Registry service*. this service has three methods associated with it and is used by the ServEnt component:
  - addServent—adds a ServEnt and its static information to the database;
  - removeServent—removes the ServEnt from the database;
  - updateStatus—updates the status (i.e. processors available) for a ServEnt.
- *ResourceRequest service*. This has one method and is used by the Initiator:
  - resourceRequest—given a set of job parameters, this returns a list of available resources which match or exceed the parameters.

The Locator is hosted within a Tomcat [11] server, and utilizes the Axis [12] framework for constructing SOAP [13] messages and Web services. The latter provides a convenient model for accessing the Locator service, where performance is not paramount; however, general inter-machine communication in JPie is conducted directly and efficiently over TCP.

3.1.2. ServEnt

The ServEnt acts as the access level for the ILS model to utilize the underlying resource. When the ServEnt is installed on a machine, the user must supply the architectural information (based on XRSL [14]) that is in turn passed on to the Locator. It is also intended that a simple, small benchmark code be run to determine how the performance compares to other resources. This benchmark figure will be used in the algorithm for selecting appropriate resources at the Locator. The ServEnt also can accept JPie tasks from an Initiator, and will pass metrics of the task run to the Initiator when it has completed.

DOI: 10.1002/cpe

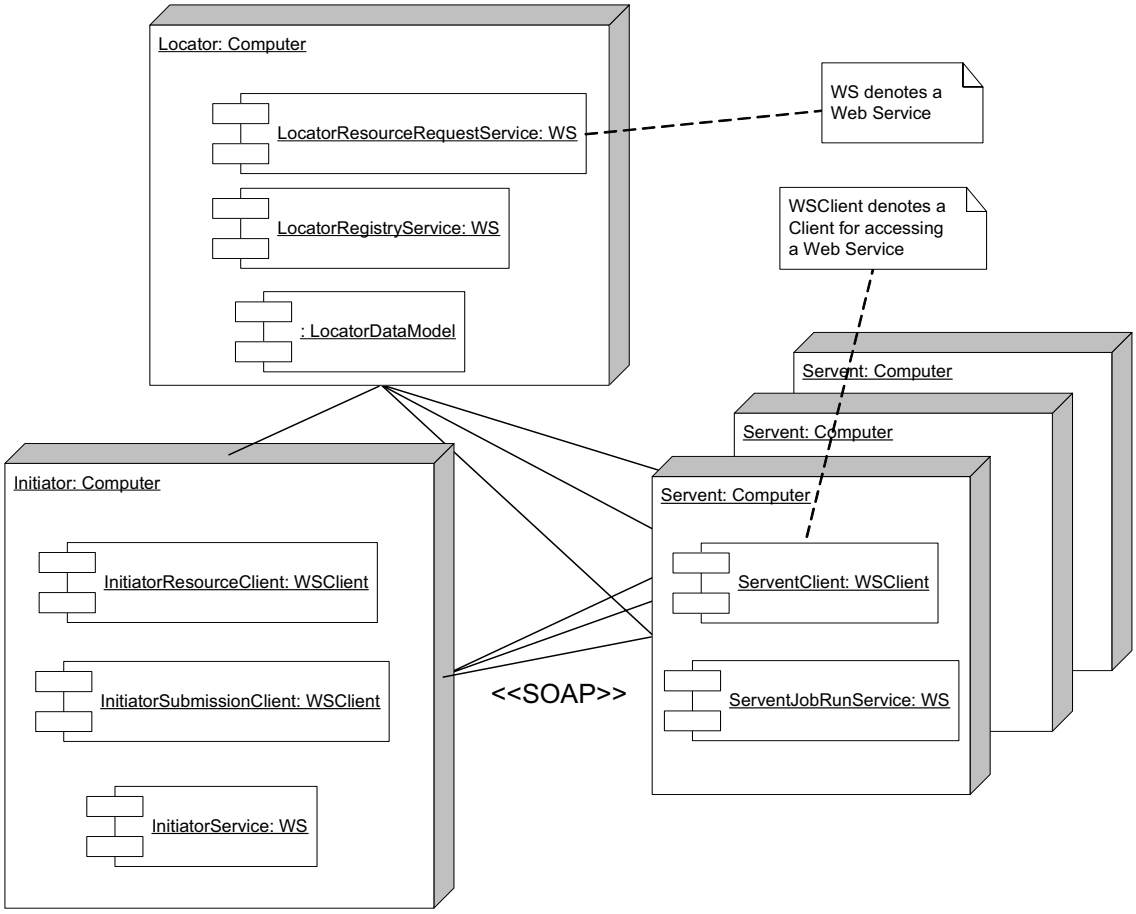


Figure 4. Deployment diagram for the ILS model.

3.1.3. Initiator

The Initiator instigates JPie tasks. Given the address of a Locator and a job description, it requests resources from the Locator and receives the addresses of the resources available and appropriate to the job. Note that the number of resources returned can be set at the Locator to match exactly the number requested, or to exceed it. This is because at the next step the Initiator negotiates with the resources in turn via their ServEnts on whether they can accept the job. If declined by one resource, then the next ServEnt is negotiated with. This extended list prevents the need for repeated contacts with the Locator. Once the negotiations have completed, the Initiator submits the task to the ServEnt.

DOI: 10.1002/cpe

3.1.4. Methodology

There are several technologies already in existence that offer similar functionality to the interactions described above, and a survey of these was undertaken as part of the PGPGrid project. However, none of the technologies matched the requirements exactly, and those that came closest were relatively heavyweight solutions, which would have required significant investment in terms of time and effort in adapting them to our requirements. Due to the inherent difficulties of such an approach, and given the time constraints, it was decided to utilize the Web services paradigm. This allowed an iterative approach to developing the functionality, and the resulting implementation should also be straightforwardly extensible to the emerging WSRF [15] standards.

4. MODEL CONFORMATION

One of the project goals is to demonstrate the use of dynamic 3D data to drive computer-generated animation. In addition to providing a more productive workflow we hope that animation generated in this way will prove to be much more believable than animation generated by traditional means. In particular, we emphasize the subtle and complex facial movements that we can capture, which are often lacking in artificially generated animations.

The model-building process produces a set of discreet triangulated meshes, one per frame. We now describe the workflow developed to move from these discreet models to a single animated model suitable for inclusion in a professional studio production. The normal process of animation would require an animator to systematically deform key frames of a sequence by hand. In-betweening would then be used to generate the intermediate positions. In contrast to this, our approach is to use a semi-automatic process whereby a person brings a few key points of a frame into alignment with the captured model. Other points are then automatically brought into alignment. In this way a facial animation requires between one and two minutes of human intervention per key frame, about a quarter of what would be required in conventional work. In our case we have higher realism, as not only are the landmark points brought into correspondence with the actor's face, but so are hundreds of non-landmark points across the surface of the skin.

4.1. Model mark-up

The most important step is moving from a set of discreet models to a single unified model that an animator can interact with consistently across frames. This is achieved by mapping a generic model to the captured data using a process called conformation [16]. Before we conform the generic model we must first identify corresponding feature points between it and the captured data.

This relies on tracking the position of features we call landmarks. These landmarks are points in the captured data corresponding to features that must be mapped accurately in order to preserve the quality of the animation.

After experimenting with several different sets of landmark points we have opted to use a subset of the points defined in the MPEG-4 standard for facial animation [17]. However, it was necessary to introduce additional landmarks to stabilize the structure of the mesh between frames (see Figure 5).

DOI: 10.1002/cpe

Figure 5. Landmarks in position on a generic mesh.

The task of recording the position of these landmarks is complicated by the volume of data. Every frame consists of one VRML model and four texture images, each of which is 24-bit colour at a resolution of 640×480 pixels. Taking into account the data structures necessary to display and manipulate this data interactively, every frame can use up to 6 MB of memory (dependent on mesh complexity). A simple scaling calculation shows that an eight second sequence at 25 frames per second (FPS) requires at least 1 GB of memory.

A mark-up interface has been developed that allows the user to operate on longer sequences in one session. This interface enables the user to select any VRML model as the 'base' position for the landmarks. This approach has several advantages:

- mesh structure can help the user identify which landmark they are interacting with;
- a 'base' position that is close to the underlying capture data reduces the amount of work required;
- the landmarks used can be changed simply by selecting a different initial mesh.

Once the mesh containing the landmarks has loaded the user can select and drag landmarks across the surface of the model. When all landmarks are in place the user moves to the next frame. Duplicated effort is minimized by projecting landmarks from the previous frame onto the model in the new frame, thus only minor corrections are needed between frames.

To reduce the demands of this process on the computer being used, partial sequences can be marked-up. The last frame from one partial sequence is loaded as the start point for the next sequence, enabling the user to continue with minimum interruption. In this way it takes three minutes to position landmarks for a single frame. Due to memory limitations, working with one second at a time results in the best trade-off between user effort and software performance.

DOI: 10.1002/cpe

4.2. Conformation

Once the landmark points have been recorded, then the generic model can be conformed to the captured data. The conformation algorithm, which deforms a generic model to scanned data, comprises a two-step process: global mapping and local deformation.

- *Global mapping*. Global registration and deformation are achieved by means of a 3D mapping based on corresponding points on the generic model and the scanned data. The 3D mapping transforms the landmark points on the generic model to the exact locations of their counterparts on the scanned data. All other points on the generic model are interpolated by the 3D mapping; this mapping is established using corresponding feature points through radial basis functions [18]. The global mapping results in the mesh of the generic model being subject to a rigid body transformation and then a global deformation to become approximately aligned with the scanned data.
- *Local deformation*. Following global mapping, the generic model is further deformed locally, based on the closest points between the surfaces of the generic model and the scanned data. The polygons of the generic model are displaced towards their closest positions on the surface of the scanned data.

An elastic model is introduced in the second step of the conformation. The global deformed mesh is regarded as a set of point masses connected by springs. During the second stage of the conformation process, the mesh anchored to landmarks is relaxed to minimize its internal elastic energy. The displacements of the vertices deformed to their closest scanned surface are constrained to minimize the elastic energy of the mass-spring system. The governing equation for the relaxation of a single vertex $\mathbf{x}_v$ is

$$m_v \ddot{\mathbf{x}}_v + d_v \dot{\mathbf{x}}_v + \sum_{j=1}^{l} \mathbf{f}_{vj} = \mathbf{f}_v^{\text{ext}} \tag{1}$$

where m_v is the mass of the vth vertex, d_v its damp factor, $\mathbf{f}_{vj}$ is the internal elastic force of the spring connecting the jth vertex to the vth vertex, l is the number of vertices connected to the vth vertex and $\mathbf{f}_v^{\text{ext}}$ is the sum of the external forces on the vth vertex.

The relaxation uses the following procedure.

(1) Each vertex $\mathbf{x}_v$ is deformed to its closest position on the measured shape. Conformation ends here if no elastic constraint is applied; otherwise, continue to the next step.
(2) The sum of the internal forces $\sum \mathbf{f}_{vj}$ on each vertex $\mathbf{x}_v$ is calculated. There is no external force in this model.
(3) The acceleration, velocity and position of each vertex $\mathbf{x}_v$ are updated with the exception of the vertices of the triangles of the mesh in which the landmarks lie.
(4) Steps (1) to (3) are repeated until the mesh is settled or the number of iterations exceeds a fixed number.

We have found that better results at this stage can be obtained by conforming the generic model to the first frame of the captured sequence and by then conforming the resulting mesh to each subsequent frame. This approach means that the deformation required is relatively small and is hence less prone to introduce errors or noise.

DOI: 10.1002/cpe

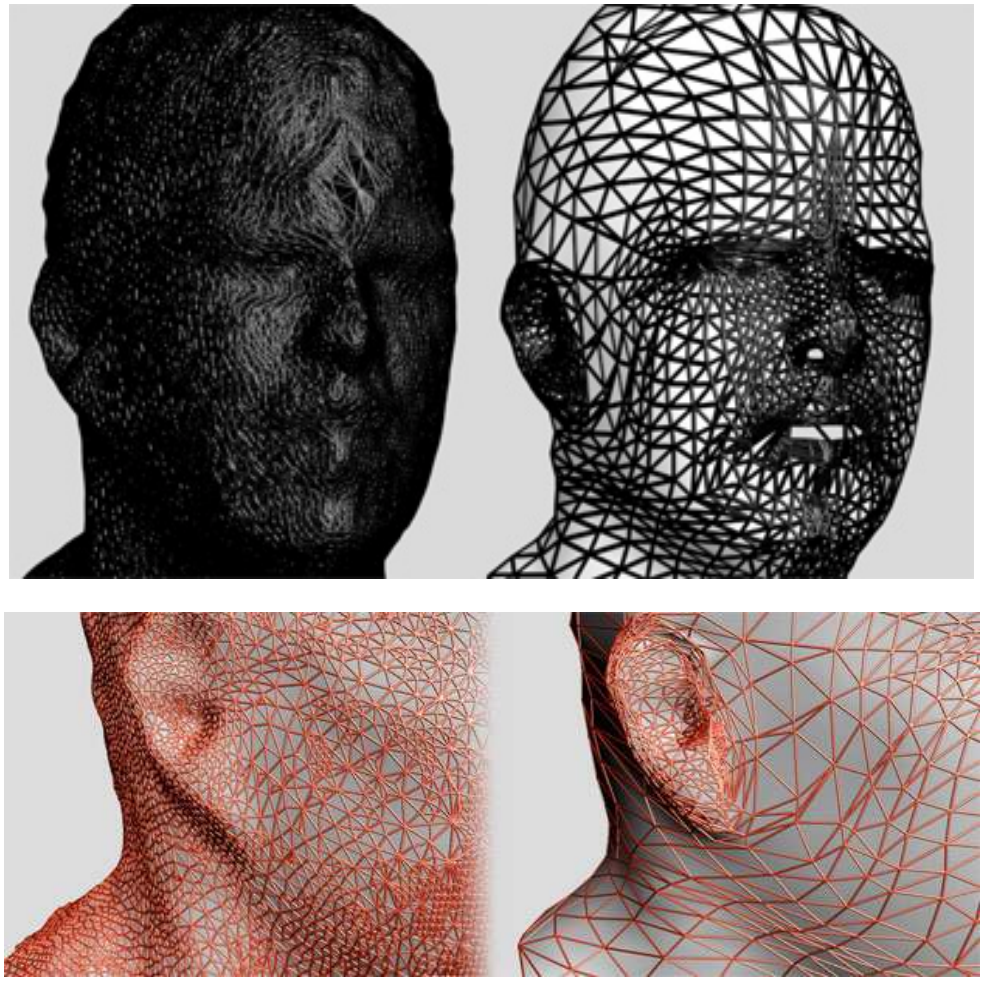


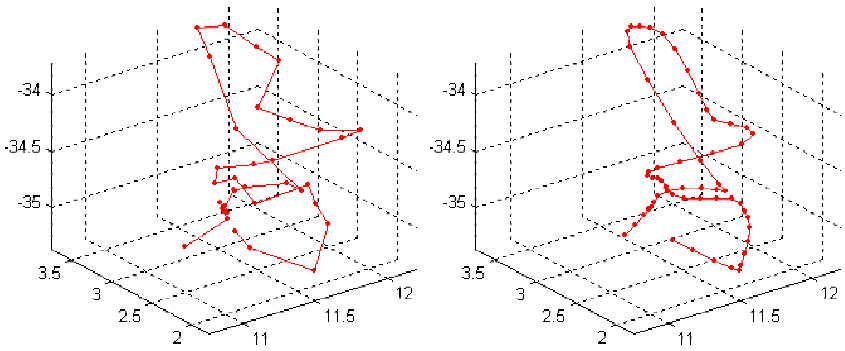
Figure 6. Unconformed versus conformed mesh structures. Note the reduction in the number of polygons, the move to anatomically relevant polygons and the increase in effective detail around the ear obtained by convolving through time.

4.3. 3D Studio Max animation

After conformation we have a set of discreet models with the shape and motion from the scanned data but having the mesh structure of the generic model. In this form the sequence is not directly usable by an animator; however, the transformation to a single model with a set of key frames defining the motion is straightforward.

Even after conformation there is still some noise present in the data that causes it to stand out when incorporated into 'clean' computer-generated environments. To combat this we apply a high-frequency

DOI: 10.1002/cpe



(a) (b)

Figure 7. Trajectory of a vertex (a) before and (b) after filtering.

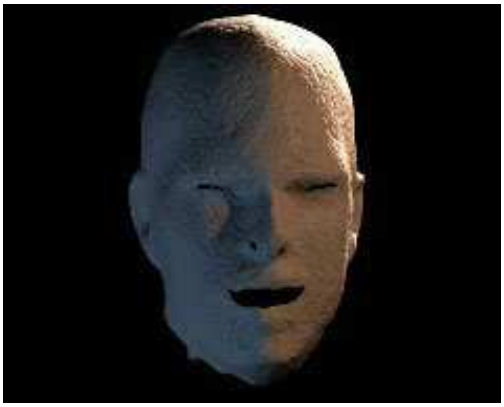


Figure 8. Image from test sequence rendered by Pepper's Ghost.

filter to the motion of individual vertices. This filtering is achieved by convolving the trajectory of each vertex with a symmetric one-dimensional Gaussian kernel of the form

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp -\left(\frac{x^2}{2\sigma^2}\right)$$

In areas of high detail, the quality of the mesh is improved as a result of considering spatial information from neighbouring frames (see Figures 6 and 7) and movements resulting from noise are eliminated or reduced. To ensure that no important details are sacrificed, an interface has been incorporated into 3D Studio Max allowing the degree of filtering to be specified as the animation is being imported, thus allowing the animator full control over the extent of the filtering.

Early indications are that this workflow will produce very good results, maintaining a good balance between realism of motion and quality of the model (see Figure 8).

DOI: 10.1002/cpe

5. CONCLUSIONS

The parallelization experiments described above show that the 3D scanner application is extremely well suited for distribution in a Grid-type environment. With sufficient resources, parallelization 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 π-calculus that satisfies these requirements. This has been successfully implemented in a preliminary form and work is proceeding to integrate it fully with the Grid protocols.

Using the technology described here PGP have been able to produce a short demonstration video that uses the entire production chain from 3D capture, through body conformation to final rendering using the parallel facilities offered by the EPCC. Experience with the production of this demonstrator leads us to believe that the techniques pioneered in this project could considerably reduce the implementation costs of large animation projects.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the funding by NESC, Edinburgh, U.K. and thank our project partners Peppers Ghost Productions for their input.

REFERENCES

1. Cockshott WP, Hoff S, Nebel J-C. Experimental 3-D TV studio. *IEE Proceedings—Vision, Image and Signal Processing* 2003; **150**(1):28–33.
2. Zhengping J. On the multiscale iconic representation for low-level computer vision systems. *PhD Thesis*, Turing Institute and the University of Strathclyde, 1988.
3. Urquhart CW. The active stereo probe, the design and implementation of an active videometrics system. *PhD Thesis*, Turing Institute and the University of Glasgow, 1997.
4. Lorensen WE, Cline HE. Marching cubes: A high resolution 3D surface construction algorithm. *ACM Computer Graphics* 1987; **21**(4):163–169.
5. Milner R. The polyadic π-calculus: A tutorial. *Technical Report ECS-LFCS-91-180*, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, U.K., October 1991.
6. Milner R, Parrow J, Walker D. A calculus of mobile processes. *Information and Computation* 1992; **100**:1–77.
7. Ferrari A. JPVM: Network Parallel Computing in Java. *Proceedings of the ACM Workshop on Java for High Performance Network Computing*. ACM Press: New York, 1998.
8. Gray P, Sunderam V. IceT: Distributed computing and Java. *Concurrency: Practice and Experience* 1997; **9**(11):1161–1167.
9. Martin JMR, Welch PH. A CSP model for Java Multithreading. *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000; 114–122.
10. Web services. http://www.w3.org/2002/ws/ [June 2005].
11. Apache Tomcat. http://jakarta.apache.org/tomcat/ [June 2005].
12. Axis. http://ws.apache.org/axis/ [June 2005].
13. Simple Object Access Protocol. http://www.w3.org/TR/soap/ [June 2005].
14. Extended Resource Specification Language. http://progress.psnc.pl/English/architektura/xrsl.html [June 2005].
15. WS-Resource Framework: FAQ. http://www.globus.org/wsrf/faq.asp [June 2005].
16. Ju X, Siebert P. Conformation from generic animatable models to 3D scanned data. *Proceedings of the 6th Numérisation 3D/Scanning 2001 Congress*, Paris, France, 2001; 239–244.
17. Pandzic IS, Forchheimer R. *MPEG-4 Facial Animation*. Wiley: New York.
18. Ju X, Siebert P. Individualising human animation models. *Proceedings of Eurographics 2001*, Manchester, U.K. Blackwell: Oxford, 2001.

DOI: 10.1002/cpe