The portable UNIX programming system (PUPS) and CANTOR: a computational environment for dynamical representation and analysis of complex neurobiological data

M. A. O'Neill¹* and C. C. Hilgetag²

¹*Bee Systematics and Biology Unit, Hope Entomological Collections, Parks Road, Oxford OX1 3PW, UK*

²*Department of Anatomy and Neurobiology, Boston University School of Medicine, 700 Albany Street W746, Boston, MA 02118, USA*

Many problems in analytical biology, such as the classification of organisms, the modelling of macromolecules, or the structural analysis of metabolic or neural networks, involve complex relational data. Here, we describe a software environment, the portable UNIX programming system (PUPS), which has been developed to allow efficient computational representation and analysis of such data. The system can also be used as a general development tool for database and classification applications. As the complexity of analytical biology problems may lead to computation times of several days or weeks even on powerful computer hardware, the PUPS environment gives support for persistent computations by providing mechanisms for dynamic interaction and homeostatic protection of processes. Biological objects and their interrelations are also represented in a homeostatic way in PUPS. Object relationships are maintained and updated by the objects themselves, thus providing a flexible, scalable and current data representation. Based on the PUPS environment, we have developed an optimization package, CANTOR, which can be applied to a wide range of relational data and which has been employed in different analyses of neuro-anatomical connectivity. The CANTOR package makes use of the PUPS system features by modifying candidate arrangements of objects within the system's database. This restructuring is carried out via optimization algorithms that are based on user-defined cost functions, thus providing flexible and powerful tools for the structural analysis of the database content. The use of stochastic optimization also enables the CANTOR system to deal effectively with incomplete and inconsistent data. Prototypical forms of PUPS and CANTOR have been coded and used successfully in the analysis of anatomical and functional mammalian brain connectivity, involving complex and inconsistent experimental data. In addition, PUPS has been used for solving multivariate engineering optimization problems and to implement the digital identification system (DAISY), a system for the automated classification of biological objects. PUPS is implemented in ANSI-C under the POSIX.1 standard and is to a great extent architecture- and operating-system independent. The software is supported by systems libraries that allow multi-threading (the concurrent processing of several database operations), as well as the distribution of the dynamic data objects and library operations over clusters of computers. These attributes make the system easily scalable, and in principle allow the representation and analysis of arbitrarily large sets of relational data. PUPS and CANTOR are freely distributed (<http://www.pups.org.uk>) as open-source software under the GNU license agreement.

Keywords: persistent computations; homeostasis; relational data; connectivity analysis; cluster computation; parallel computation

1. INTRODUCTION

Biological data are typically complex and often possess features that hamper straightforward quantitative analysis. This problem applies equally to morphological descriptions of biological structure, brain activity data from neuroinformatics databases, taxonomic character data from computer aided taxonomy (CAT) databases, or experimental data required to mathematically model

ecosystems. The difficulties in analysis are exemplified through data about structural and functional aspects of neural systems that are now becoming available in systematically collated databases (e.g. Burns, this issue; Stephan *et al.*, this issue). Neuroinformatic data are often incomplete, depend on subjective evaluation (e.g. Burns & Hilgetag 2000) and are, hence, not always completely reliable. The data may even contradict themselves within subcomponents of the overall dataset. Additionally, and in accord with the neural systems that they describe, these data are characterized by multiple, 'tangled',

*Author for correspondence (mao@ermine.ox.ac.uk).

Figure 1. A connection map of visual cortical areas in the macaque monkey that displays the properties of a tangled dataset. The diagram was derived by randomly arranging areas in two dimensions and linking them as indicated by the connectivity data in figure 3. Links stand for either uni- or bidirectional connections. This diagram and figure 3 probably contain valuable information about the primate visual system; however, the exact organization of this connectivity network is far from obvious.

interrelationships, which often necessitate analysis within the context of the global dataset. Figures 1 and 3 provide a glimpse of one example for such data.

In tangled datasets, individual items of data often exhibit complex static and dynamic relationships with other data items, giving rise to a class of data objects the properties of which are very different from those that are routinely represented in classical (e.g. structured query language (SQL)) databases. In particular, complex interrelationships between data items mean that it is very inefficient to transform them into a flat file format, which is a standard technique when storing data in a conventional relational database. To deal effectively with complex biological problems, data must be represented in a way that encapsulates the high-dimensional interrelationships between the individual items of data. As a consequence, computational analysis of complex biological data on serial computers, such as commodity personal computer (PC) and workstation hardware, requires a special kind of support environment to be both effective and timely. Two main factors have to be taken into account.

- (i) Data complexity, expressed, for instance, by tangled relationships between data items, necessitates purpose-designed computational approaches.
- (ii) The extended run times needed for the analysis of complex data require applications to be persistent.

One way of achieving this stability is through a 'persistent object store', which permits data to be mapped directly from a file into the address space (i.e. the working memory) of the application process that requires the data. In a networked or parallel computational environment, it may also be desirable to simulta-

neously map the persistent object store into the address spaces of many cooperating application processes, in order to distribute the computational workload or to assess many aspects of a complex dataset at the same time. Such distributed design has several implications for the structure of the computational environment. For instance, careful consideration must be given to semaphores in order to avoid race conditions in which the persistent object store is multiply accessed. Additionally, if an application fails to complete a transaction, the store must be able to restore itself (atomically) to a well-defined state, e.g. the state prior to the unsuccessful transaction.

In the following, we describe a computational environment, the portable UNIX programming system (PUPS), that has been specifically designed to deal with the properties of complex biological, and in particular neuro-informatic, data. We outline the philosophy of this approach in § 2 of the present paper. In § 3, we explain how the environment has been implemented on different computer architectures, and we demonstrate the successful application of this environment in the representation and analysis of biological data in § 4. Finally, we discuss how our approach compares with alternative computational solutions.

2. CONCEPTS OF THE PUPS ENVIRONMENT

(a) *Homeostasis*

Homeostasis is a theme that is highly pertinent to the ideal attributes of an environment that supports persistent and complex biological computations. Homeostasis is the tendency of a system to seek stability by opposing any external influences that may try to alter its state. It is a key property of systems adapted to persist in an unstable and potentially hostile environment, such as living biological systems or processes within a time-sharing cluster of computers. PUPS employs several homeostatic mechanisms, including the migration and protection routines ideally possessed by persistent computational processes, and the self-repairing (atomic) characteristics of an idealized persistent object store.

(b) *Computational persistence*

Computational persistence is a required property of systems that represent or analyse complex datasets. The intricacy of biological data, in particular the fuzzy relationships that may frequently link individual data items, lead to systems with complex dynamic properties. This means that even with relatively powerful parallel or network (cluster) computing facilities, applications that deal with complex biological data may have to run for a period of days to weeks. This has a number of implications.

- (i) Applications must be reliable: they must, for example, be re-startable in the event that the host that is running them crashes.
- (ii) The computational effort of processing biological data may be significant. Therefore, in network and parallel computer environments, the provision of some form of dynamic load balancing mechanism is desirable. Such a mechanism permits the automatic migration of application processes to those nodes within the network that are optimally suited to

supporting the computational payload of a given application. In addition, automatic process migration also facilitates reliability because it permits application processes to migrate from nodes that are malfunctioning (due to hardware faults, impending power supply failure, etc.).

- (iii) Applications that process biological data should be capable of using computational resources in an adaptive manner. Given the lengthy application process lifetimes of days to weeks, applications that monitor their execution environment are likely to be more computationally efficient than processes that are not capable of adaptive resource management. In particular, process level resource management means that long-running processes are able to protect objects (e.g. files, communications channels and threads of execution) that are vital to the process or its computational payload from undesirable external influences, for example, the accidental or malicious deletion of files. Processes that manage their own resources are also at an advantage when competing against other programs for resources such as memory and central processing unit (CPU) cycles. For example, a process that requires significant memory resources to run can capitalize on the demise of other (badly written) programs by waiting for such programs to crash and then dynamically appropriating the resources they release. Another area where adaptive resource management may be advantageous is the control of output data. Neurological and other bioinformatic applications typically produce a great deal of data. In a network computer environment, disk space may be yet another scarce resource. Applications that can react appropriately when the disk to which they are writing fills up or, worse still, fails are likely to have a selective advantage.
- (iv) Users are understandably worried about whether their application, which may have been running for a considerable amount of time, is malfunctioning. Environments that support persistent computations should therefore provide mechanisms that allow the user to dynamically interrogate their applications in order to assure themselves that they are still working correctly (after, for instance, several CPU-weeks of computation).
- (v) Given the kinds of data analysis appropriate for analysing complex biological data (for example, stochastic optimization and fuzzy template matching), many applications would benefit if their operational parameters and indeed their code structure could be modified and adapted to the features of the data under investigation while they are running. Persistent computations therefore would be facilitated if mechanisms exist for dynamically steering the parameters and functionality of the program.

3. IMPLEMENTATION

(a) *Overview of the portable UNIX programming system*

PUPS, represents an attempt to produce a standard environment for the support of complex and persistent

computations. The system implements many of the features discussed in the previous section. PUPS supports homeostatic mechanisms including:

- (i) Automated process migration via MOSIX, an extension to LINUX, developed at the Hebrew University of Jerusalem (Barak *et al.* 1993).
- (ii) Support for checkpointable (e.g. recoverable) processes via CONDOR, a high-throughput parallel computer environment developed at the University of Wisconsin (Litzkow *et al.* 1988), and the University of Tennessee Checkpointing Library (*libckpt*) (Plank *et al.* 1995). Note both of these libraries require modification so that they can deal with FIFOs and redirected input–output (I/O) in a sensible manner.
- (iii) Homeostatic protection of objects, such as files, FIFOs (first in first outs), etc.
- (iv) Peer-to-peer and user-to-peer dynamic interaction with running processes.
- (v) Support for dynamic goal reassignment and steering parameter update in running applications.
- (vi) Support for manipulation and storage of complex biological datasets via a practical multi-host/multi-process implementation of a persistent object store.

The PUPS system is coded in ANSI-C. Although it has been developed under the LINUX operating system using the Free Software Foundation GNU compiler tools, PUPS has been ported to a number of POSIX.1b compliant operating systems, including OSF1, Solaris, SunOS 4.1 and BSD 4.4. POSIX is a standard defined by the Institute of Electrical and Electronics Engineers (IEEE), which defines programming and system tool interfaces for UNIX-like operating systems. With relatively little effort, it could also be ported to the Microsoft Windows NT and Windows 2000 environments as both these PC operating systems offer POSIX.1b compliance.

(b) *Structure of the PUPS environment*

The PUPS environment can be divided into three parts: a set of core C libraries; a set of daemons, to support network and parallel computations; and a set of standard applications, the PUPS service functions.

(i) *Core libraries*

There are five core libraries in the standard PUPS distribution. These are linked with computational payload code in order to produce a PUPS application binary. The standard libraries are as follows.

utilib

Contains general purpose utilities including command tail decoding, extended I/O functionality, application process name overloading (enabling applications to be tracked by name rather than process identifier (PID) within a POSIX.1b environment), extended asynchronous event handling and a number of standard homeostats for protecting files and interprocess communication channels. (Asynchronous events (signals) are used by one process to tell another that it wants its attention. They are also used by the operating system to get the attention of application processes.)

casino

Provides a variety of pseudorandom number generators (after Knuth 1984), and generators for Poisson, normal,

binomial and other well-known distribution functions in order to support applications that use stochastic optimization (e.g. CANTOR, see §4c).

netlib

Provides support for network computing, including asynchronous signalling of processes running on remote hosts, remote execution of processes, gathering of loading and other status information from remote hosts.

psrplib

Implements process status request protocol (PSRP), a novel Internet protocol enabling both users and peer processes (PSRP clients) to interact asynchronously with a running application (the PSRP server) on an Internet-wide basis, as illustrated in figure 2. PSRP is a secure protocol. With the provision of appropriate authentication tokens (e.g. passwords or encrypted signatures) it is possible for a PSRP client to access and make use of resources that it does not own. (This facility was provided to enable effective resource sharing for multi-person, multi-site projects, as it permits seamless access to an instance of key PSRP applications, such as CANTOR, by many users, irrespective of who actually owns that instance.) In addition, the PSRP protocol is multi-threaded, which means it is possible for a running PSRP server application to be simultaneously accessed by multiple PSRP clients. The PSRP library also provides a large number of inbuilt functions that support standard PSRP client-server interactions including:

- (i) Status information about the PSRP server, including the number and identity of any connected clients, memory use, attached dynamic objects (e.g. persistent object stores and/or dynamic functions), number of open files, etc.
- (ii) A mechanism for dynamically importing dynamic functions into a running application (the PSRP interface supports the dynamic linking of code modules with strong type checking and possible type broking such as the promotion of *int* to *long* or the deprecation of *double* to *float* using the POSIX.1b dynamic link-loader).
- (iii) A mechanism for attaching and modifying persistent object stores: these are managed using the POSIX.1b memory mapping interface, which permits a process to map directly the contents of a file (containing the persistent object store) into its address space. The POSIX.1b memory mapping functionality has been extended to enable memory mapping over a network file system (NFS). NFS is a way of sharing files in a distributed computing environment.
- (iv) Mechanisms for modifying the homeostatic responses of an active PSRP server (e.g. server checkpointing, migration and homeostatic protection can be switched on or off by appropriately authenticated PSRP clients).

mvmlib

Provides support for PUPS processes to optimally manage their own memory. For example, this mechanism is used by the CANTOR application (see §4c) and permits it to deal efficiently with huge rule matrices.

A full list of built-in PSRP server commands is shown in table A1 of the Appendix.

Figure 2. Schematic showing a process status request protocol (PSRP) server process connected to one local and one remote PSRP client process.

(ii) *PUPS/PSRP daemons*

The PUPS daemons facilitate the transparent network and parallel computing features of PUPS, including signal relaying to remote hosts, remote process execution, initializing remote PSRP client sessions and disseminating host kernel loading statistics. The PUPS daemons use a novel protocol to check the identity of clients making requests for daemon services and to propagate authentication information to PUPS applications (e.g. PSRP clients and servers) that are running on remote hosts. There are five PUPS/PSRP daemons as follows.

rttyd

Makes a connection between a remote PSRP client running on a remote host and a PSRP server running on the daemon's local host (subject to authentication) via a full duplex terminal interface.

xrshd

Makes a connection between a remote PSRP client running on a remote host and a PSRP server running on the daemon's local host (subject to authentication) via a half duplex BSD socket interface. Sockets are a form of communication channel that permit reliable (e.g. stream based) communications between a pair of host computers on the Internet.

rfifod

Permits FIFOs and other non-regular file system objects to be exported via NFS.

rpsd, rloadd, rupd

Disseminate information about the local host's current computational load and any PSRP server processes that it may be running.

xkilld

Relays signals sent by processes running on remote hosts to target processes that are running on *xkilld*'s local host.

(iii) *PUPS service functions*

The PUPS service functions are a set of tools that facilitate the construction of user applications in the PUPS environment. Many of the functions (e.g. *fsw*, *xcat* and *xtee*) are used to provide specific services (e.g. file-system monitoring, foreground monitoring tool insertion points and stream splitting, respectively) within virtual dataflow machines (VDM), which consist of cooperating pipelines of application processes. Other functions (e.g. *nkill* and *xrshd*) extend the functionality of existing UNIX tools (e.g. *kill* and *rsh*) in the context of the PUPS environment. PUPS currently provides seven service functions.

nkill

Permits processes, which may be remote, to be signalled to using a unique name, rather than their PID.

xrsh

Uses the *xrshd* PUPS daemon to start application processes running on remote hosts. *Xrsh* is intended to be a replacement for the BSD *rsh* command; it is more secure than *rsh* and also incorporates support for static load balancing and execution of multi-host process pipelines and VDMs.

fsw, *xcat*, *xtee*

Permit the efficient implementation of VDMs. *Fsw* is a file system watcher that is able to halt the execution of any VDM of which it is a member in the event of the VDM output file system becoming full. (Alternatively, *fsw* can try and move output files to an alternative file system that is not yet full, and then resume execution of its VDM.) *Xcat* is primarily intended to act as a place marker within VDMs that can be over-forked by ephemeral processes, e.g. X and other interactive applications. (X is a public domain network-capable windowing environment developed at MIT that is a popular graphical front end for UNIX-like operating systems.) Typically, *xcat* processes provide targets within VDMs that can be overlaid by interactive monitoring and/or visualization applications.

embryo

This is an uncommitted PSRP server application. Typically *embryo* is used to test new PUPS/PSRP installations, but it may also be used to build applications the computational payloads of which are constructed of dynamic functions.

psrp

Provides a text-based client that allows the user at an interactive terminal to communicate with PSRP server processes. From the standpoint of the user, *psrp* appears to be a pseudo-shell that supports many of the features associated with true UNIX shells, such as command history, command line editing, macro language, initialization scripts, etc. With the *psrp* client the user can open an asynchronous connection to any PSRP server to which they authenticate themselves. Once a communication channel exists between the *psrp* client and the server, the PSRP protocol is used to transfer requests to the server and its responses back to the *psrp* client and hence to the user.

(c) *Practical implementation issues*

The PUPS software and several applications built on the PUPS environment, most importantly the CANTOR package described in §4c, are freely available as open-source software under the GNU license agreement at

<http://www.pups.org.uk>. This Web site also provides detailed manual pages, instructions for software installation and configuration, as well as example code demonstrating the usage of PUPS features. The site is further intended to serve as a forum for potential users of the environment, and it will provide updates on the latest code developments and applications. The PUPS environment supports, and indeed encourages object-orientated design strategies and (dynamically) reusable code. The system is currently available for Solaris and LINUX systems, but it is highly portable and should run with few changes on any UNIX or UNIX clone that complies with the IEEE POSIX.1b standard.

The low-level (PUPS) components of the system have been extensively tested using the image-to-ground (ITG) stereo matching package, and the CANTOR and SERIATE (a bit-vector seriation tool) neuroinformatic applications. This testing process, which involved many weeks of continuous operation under production conditions, identified and eradicated many subtle bugs in the system. Potential users are also encouraged to test the code and provide bug feedback to <http://www.pups.org.uk>.

At the time of writing, the interfaces to the system, in terms of both the application programmer interface (API) for interfacing database systems and the actual user interface, remain relatively primitive. Work is being undertaken to remedy these deficiencies by groups at the University of Southern California and by Ecsponent, a software house specializing in open-source software. Ecsponent may also provide ongoing support and maintenance for the PUPS/PSRP environment, in the form of an alternative Web site that distributes code and provides enhanced system documentation. Several commercial applications, which require user-friendly graphical interfaces, are currently being built using the PUPS/PSRP environment. As a result of these developments, it is likely that graphical user interface (GUI)-based interface tools for PUPS/PSRP, implemented using either the *Python*, *Qt* or *Tk* toolkits, will be available in the near future.

4. EXAMPLE APPLICATIONS BASED ON PUPS

(a) *Implementation of PUPS applications*

Building applications using the PUPS system is straightforward. Currently, this can be accomplished in two ways.

- (i) Applications written in C or C++ may be linked at compilation time with functions in the PUPS libraries in the traditional, static manner. For a process to run as a PUPS/PSRP application, the application writer must call a number of standard library functions in order to correctly set up the PUPS/PSRP environment for the process.
- (ii) Dynamic functions may be written (in C or C++), compiled as dynamically loadable modules and then linked with a running program using PSRP dynamic loading facilities, via a PSRP client that is attached to the target process. Some example code is provided in the Appendix, table A2.

Although the PUPS system has been designed to support any persistent computation, it offers a number of facilities that are specifically aimed at tangled databases,

which often arise in bio- and neuroinformatic problems. With an eye to the future, the most significant of these are the support PUPS provides for the stochastic manipulation of complex databases, for example, making it possible to efficiently mine databases and optimally for information and the persistent object store, which allows bio- and neuroinformatic database applications to be written efficiently for both parallel and network-cluster computational environments.

The PUPS environment has already been used to build a variety of applications that include:

- (i) The ITG stereo ranging system (Denos & O'Neill 1992, 1996);
- (ii) Neuroinformatic applications using the optimization network processor CANTOR for hierarchical and cluster network analyses (Hilgetag *et al.* 1996, 2000*a,b*; Scannell *et al.* 1999; Stephan *et al.* 2000) and the bit-vector seriation tool SERIATE (Young *et al.* 1995);
- (iii) Further bioinformatic applications including CAT systems, such as the digital identification system (DAISY) (Weeks *et al.* 1998) and enhanced versions of the SERIATE tool that are capable of seriating arbitrary forms of data (e.g. character sequences and numeric vectors) in addition to bit-vectors (Bridge *et al.* 2000).

(b) The ITG stereo ranging system

Historically, ITG, which is an area correlation stereo matcher based on the work of Gruen (1985) and Otto & Chau (1989), was the first major application to be implemented under the PUPS environment. Many features of the PUPS environment, notably the PSRP client-server protocol and the process-centred memory management features offered by the PUPS core library *mwmlib*, were developed in order to support it. ITG is a tool for constructing depth-maps from optical stereo imagery. It is used both as a well-known test-bed application for ongoing development of the PUPS environment, and to support research projects in areas as diverse as the earth sciences (looking at modification of drainage in the neighbourhood of active faults) and behavioural ecology (e.g. elucidating the mechanisms of insect flight, Willmott *et al.* 2001).

(c) Neuroinformatic applications using CANTOR

The PUPS environment has been extensively tested and employed in analyses of neural connectivity. Connectivity data, such as the ones shown in figure 3 for interconnections of cortical areas in primate visual cortex, have all the attributes of tangled, incomplete and partly unreliable datasets. As more compilations of such or similar data are going to become available in the future, it is important to devise appropriate methods for their evaluation. The following sections describe two neuroinformatic analysis approaches that use the CANTOR software implemented under PUPS.

(i) CANTOR

CANTOR is a software processor for the evaluation and restructuring of object networks. The processor makes use of the homeostatic routines offered by the PUPS environment and adds the functionality required, for instance in the analysis of neural connectivity, such as

an evolutionary optimization algorithm that can be combined with various multi-parameter cost functions.

The networks that are being represented in CANTOR consist of objects, rather than of simple nodes. This means that each of the nodes can be associated with several object attributes. In principle, an unlimited number of attributes is possible. First, the attributes provide detailed information about an object and, second, they characterize the relationships of the object with all other objects in the network. This structure could, for instance, be used to represent a neuroanatomical database of brain regions, describing both the intrinsic features of the regions, such as cellular densities, prevalent morphologies or receptor properties, and the connections of the regions with each other. The representation can be structured further by giving attributes the form of hierarchical or order relations (e.g. object A contains object B). Object attributes can be used by CANTOR functions to evaluate object and network features and to rearrange the network accordingly.

CANTOR employs cost functions with categorical (nominal or ordinal) or metric components to evaluate the current organization of a network and to explore alternative rearrangements. For instance, the strength of interactions between nodes of a neural network may be described either metrically (e.g. for functional correlations between brain regions) or categorically (e.g. to describe different classes of anatomical interconnections between different regions); both ways can be represented and analysed in CANTOR. The cluster analyses detailed in Hilgetag *et al.* (2000*a*) give examples for either type of cost function.

The values derived in cost functions can be used by CANTOR's optimization engine to initiate rearrangements of the dataset. Because of the potential complexities of the data, the engine uses evolutionary, stochastic optimization. Briefly, the optimization starts with one or several randomly chosen initial network configurations, which are gradually improved in epochs, according to the given cost function. Rearrangement of the network proceeds through step mutations that create minimal structural changes in the network's configuration. The efficiency of the optimization is increased by looking ahead two subsequent generations of candidate networks and local minima are avoided by allowing intermediate generations with a slightly higher cost than that of their parent generation. The algorithm aims to collect all optimal solutions within an optimal cost range, the lower boundary of which is the cost of the best solution found in all previous epochs. Therefore, all candidate network solutions have to be compared with the cost and structure of already existing solutions, this operation can represent the limiting step of the algorithm for highly degenerate optimization problems (cf. Hilgetag *et al.* 2000*b*).

Combinations of different costs can be optimized, by default as the sum of the individual costs, although various methods of cost combination are possible. Object attributes as well as the different cost components can be weighted selectively, incorporating, for instance, subjective measures of reliability (Hilgetag *et al.* 2000*b*; Burns & Hilgetag 2000). A list of CANTOR functions and parameter settings, as well as a more detailed manual page, are available at <http://www.pups.org.uk>.

	V1	V2	VP	V3	PIP	V3A	V4	PO	V5	V4t	DP	VOT	LIP	VIP	MSTd	MSTl	PITd	PITv	7a	STPp	CITd	CITv	STPa	AITv	FEF	TF	46	FST	TH	AITd	
V1		^	o	^	^	^	^	^	^	o	o	o	o	o	o	^?	o	o	o	o	o	o	o	o	o	o	o	o	o	o	
V2	>		^	^	x	^	^	^	^	^	o	x	o	x	^	^	o	o	o	o	o	o	o	o	^?	o	o	^?	o	o	
VP	o	>		x		^	^	^	^	o		^			^	^	o	o	x	o	o	o	o	o	^?	^	o	^	o	o	
V3	>	>			^	^	^	^	^	o		o	^	^	^	o	o	o	o	o	o	o	o	o	^?	^	o	^	o	o	
PIP	x		x	x			x	^	x		^?					^?	o	o	x	o	o	o	o	o						o	
V3A	>	>	x	x			^	x		^?	^?				^?	^?	o	o	o	o	o	o	o	o	^?	o	o	^	o	o	
V4	>?	>	>	>	>	>			^?	^?	x		^	o	o	o	^	o	o	o	o	^	^	^	^?	^	^	^	^	o	
PO	x						o		x		^?		^	x	^	^	o	o	^?	o	o	o	o	o	^?	o		o	o	o	
V5	>	>	>	>	x	≥?	≥?	≥?		=	o		x	^	^	^	o	o	o	o	o	o	o	o	^	o	^?	^	o	o	
V4t	>		o	x			x	x	x						o	^?								^?		^				o	
DP	o	o	o	o	>	>	>	≥?	o				^?			^?	o	o	^	o	o	o	o	o	o	o	^	^	x	o	
VOT	o	x	>	o			>									^?	^?	^	^											o	
LIP	o		>	>	o	>	>	≤?	>		≥?			x	≤?	x	o	=?	x			o	o		o	^?	<	x	o	o	
VIP	o							x	x				x		x	x	o	x		o	o	o		o	x			x		o	
MSTd	o	>	>	>		>	o	>	>	>?	=		=?	=	=		^?	^?	=	^	o	o		o	o	^?		≥?		o	
MSTl	o	>	o	o	o	>	o	>	>	o	o	o					o	o	o	^	o	o		o	^			≤?		o	
PITd	o	o	o	o	o	o	>	o	o				o	o	x	o						^?	^?	^	^	^	<	x		^?	
PITv	o	o	o	o	o	>	o	o	o					x	o							^?	^?	^	^	^	<	x			
7a	o	o	o	o		o	o	>?	o		>		>		>	x	o	o		x	o	o	x		≥?	<	≤?	o	<	<	
STPp	o	o	o	o	o	o	o	o	o				o	o	>	>?		>?		>?		=	=	^		>?	x	≤?	>	x	o
CITd	o	o	o	o	o	o	>	o	o		o		o	o	o	o		>?	o					^?	^	<				^?	
CITv	o	o	o	o	o	o	>	o	o		o		o	o	o	o		>?	o					^?	^	<	<			^?	
STPa	o	o	o	o	o	o	>	o	o				o	o				>?	>					o		x	<		x	x	
AITv	o	o	o	o		o	x	o	o				o	o	o	o		>?			>?	o			<	o	o		≤?		
FEF	o							x	>?		x		>	>	>	>		x	>?	^?	^?					≤?	>			^	
TF	o	o	>	>	o	o	>	o	o		o			o	>		>?	>?	>		>	>	>				x	^?		x	
46	o	o	o	o		o					x				o		>?	x	x	x	>			x	≥?	o	○	=?	<		
FST	o	o	o	>	o	>	>	o	>	>			≥?	≥?	≤?	=	≥?	≥?	x	<	o	o		o	x	=	o			o	
TH	o	o	o	o	o	o	>	o	o				o	o	o	o		>	>	>	>	>	>			>?					
AITd	o	o	o	o	o	o	o	o	o				o	o	o	o		>?	x			>?			>?		>?	o			x

Figure 3. Connectivity data for the macaque cortical visual system, rearranged from Felleman & Van Essen (1991). Different colours and symbols represent the different types of structural links that connect these 30 cortical areas with each other. Area in the left column has {...} relation with area in the top row. ○, links that have been investigated but were found to be absent; x (and grey shading), existing connections for which no further qualifying information was available; < and >, feedforward and feedback connections, respectively; =, lateral connections; ≥ and ≤ indicate 'mixed' feedforward/lateral or feedback/lateral links, respectively. To facilitate recognition, these hierarchical categories have been additionally indicated by coloured shading: blue, feedforward connections; yellow, feedback connections; green, lateral connections. Question marks and lighter colours denote information that has been classified as less reliable by Felleman & Van Essen (1991); finally, empty slots show experimental information not yet available in this compilation. Areas MIP and MDP have been excluded because of the limited information available for them in Felleman & Van Essen (1991). These data obviously contain valuable and intriguing information about the organization of the primate visual system, but what organization do they imply? See Van Essen *et al.* (1992), Young (1992), Young *et al.* (1995), Hilgetag *et al.* (1996, 2000a,b), Jouve *et al.* (1998), and Sporns *et al.* (2000), for previous attempts to answer this question.

(ii) Cluster analyses

The question of which regions of the brain are likely to cooperate with each other is of great significance for many concepts of brain organization and function. One approach to investigating this problem is to identify clusters of brain regions that are more strongly linked with each other, by anatomical connections or functional relationships, than with other neural structures. We have used CANTOR for extensive cluster analyses of corticocortical and thalamocortical connectivity in the macaque and cat, based on compilations of anatomical (Scannell *et al.* 1999; Hilgetag *et al.* 2000a) and activity propagation (Stephan *et al.* 2000) data.

The cost function used in these analyses closely followed an intuitive concept of connectivity clusters (Young *et al.* 1995), considering that an optimal cluster grouping of the whole network would have as few as possible links between the different clusters, as well as a minimal number of absent connections within the clusters. Consequently, the cost, which had to be minimized in the analyses, was chosen as the sum of two components: (i) the number of all connections between all clusters, and (ii) the number of absent connections within all

clusters. It should be noted that CANTOR's ability to implement such explicit definitions for deriving optimal arrangements gives it an advantage over the traditional clustering approaches that control the clustering outcome more indirectly via metric or ranked distance measures and cluster linkage specifications. Where the available data explicitly distinguished between absent and unknown data (as in the example set shown in figure 3), only the connections explicitly known to be absent were considered in the analyses. For less-detailed datasets, both absent and unknown connections had to be treated as one category. In some cases, connectivity data compilations also contain additional information about the strength of existing connections. This information was then used in two alternative ways to weight the strength of the connections in the cost function. In a metric approach, the connection density (classified as '1', '2' or '3' for weak, intermediate or dense connections, respectively) was used as a direct multiplication factor for the respective connections. In an ordinal approach, on the other hand, these densities were employed to determine the cluster organization of connectivity networks in a staged way, first taking into account all the dense connections, then the intermediate and

Figure 4. Summary plot of three optimal cluster arrangements for the connectivity data shown in figure 3. The shading indicates the relative frequency by which the cortical areas appeared together in the same cluster. For instance, area VOT grouped with the cluster at the bottom in two-thirds of all solutions and with the one at the top in the remaining one-third. The analyses used the simple cluster cost function described in the text, without density- or reliability-weighting of individual connections. The solutions shown were obtained by assuming an equal weighting of the two cost components, which equates to giving equal importance to existing and absent connections between the areas. Solutions were also computed for systematic variations in the weighting of the two cost components, producing optimal cluster groupings that vary from just one cluster to many small, irreducible, dense clusters. In the arrangements shown, the number of connections between clusters ranged between 73 and 89, whereas the number of absent links within the clusters varied between 58 and 42, respectively, leading to a total cost of 131. See Hilgetag *et al.* (2000a) for a detailed description of the clusters detected in macaque and cat cortical connectivity.

finally the weak ones. The organization of the connectivity networks indicated by either approach was, however, very similar (Hilgetag *et al.* 2000a).

Optimal cluster groupings were evolved in epochs starting from 50 different randomly chosen cluster arrangements of the regions in the connectivity network. Each epoch could yield 50 solutions within 1% of the cost of the best solution; and we verified with the help of simple test cases that these default settings would produce an exhaustive round-up of the optimal configurations. Typically, however, only few optimal cluster groupings were produced in all epochs. Figure 4 presents a summary of all three optimal arrangements found for the data shown in figure 3. A dataset of this size required only a few minutes CPU-time per epoch on an average commodity PC running LINUX-based versions of PUPS and CANTOR.

The diagram indicates that the known connectivity specified the arrangement of visual cortical areas in two main clusters and one largely separate area, the primary visual cortex ('V1'). This result supports the view that primary visual cortex has a unique structure and function compared with other visual areas (Peters & Rockland 1994), and agrees with the long-assumed separation of the primate visual system into two broad processing 'streams'

(Ungerleider & Mishkin 1982; Young 1992). Although it has been debated whether these strongly interconnected groups of occipitoparietal areas (cluster towards the top of figure 4) and mostly temporal areas (cluster at bottom) support distinct functional modes of spatial and object vision, respectively (Ungerleider & Mishkin 1982), or rather are involved in more general cognitive functions such as visual action and visual perception (Milner & Goodale 1996), there can be no doubt that a significant connectional dichotomy exists in the primate visual system. For more detailed discussion, see Hilgetag *et al.* (2000a).

One problem in traditional cluster analyses is to determine the number of clusters that should be detected in a given dataset. This number often depends indirectly on the parameter settings of the analysis, such as settings for cluster core radii or neighbourhood relationships. Although the cluster picture presented by the analysis crucially depends on the specified number of clusters, this parameter frequently has to be chosen with the help of alternative analyses, or has to be selected *ad libitum*. The cluster analysis presented here circumvents this problem, as CANTOR detects optimal cluster arrangements across all possible cluster sizes. In other words, the evolutionary algorithm simultaneously determines the optimal number and optimal composition of clusters. Nevertheless, cluster

sizes in the configuration can be controlled via selective weighting of cost components (Hilgetag *et al.* 2000a). This feature, and the possibility of combining many different metric or categorical cost components, ensure that CANTOR's cluster algorithm is very versatile and make it suitable for the analysis of many different types of tangled networks, for instance traffic networks, relationship and similarity networks (as in the evaluation of nucleic-acid motifs).

(iii) Hierarchical analyses

An established approach in the analysis of cortical connectivity is hierarchical analysis (e.g. Felleman & Van Essen 1991). Anatomical criteria have been suggested to classify corticocortical projections as 'feedforward', 'feedback' or 'lateral', on the basis of the patterns of origin and termination of the projections across the cortical layers. Figure 3 shows such a classification for the data on the macaque cortical visual system presented in Felleman & Van Essen (1991, table 5). Classified as hierarchical rules, these data can be used to arrange cortical areas into global system hierarchies that violate as few as possible of the individual constraints (e.g. Maunsell & Van Essen 1983; Felleman & Van Essen 1991; Hilgetag *et al.* 2000b).

We automated hierarchical analysis with the help of CANTOR and defined a categorical cost function that counted the number of hierarchical constraints violated by any given candidate hierarchical arrangement. Before entering extensive computations, we verified that CANTOR operating with the hierarchical cost function would find all optimal solutions in several simple test problems that could also be solved by hand, such as $\{A > B, C > B, C < A\}$. When the cost was then applied to optimize the data in figure 3, hierarchical solutions were obtained that violated 11 of the individual constraints. This number is significantly smaller than the violation cost for optimal arrangements of the randomly redistributed data, indicating strict hierarchical regularities in the data. However, the solution space for this problem proved to be highly degenerate, in the sense that a very large number of optimal solutions with the same low cost were found for the used categorical cost function. Within 1000 randomly started epochs, which each could collect 30 solutions, the algorithm computed 6828 optimal hierarchies for the shown constraints, and the sustained rate of solutions found per epoch suggested that this did not yet constitute an exhaustive set of optimal solutions for these constraints. For a slightly different, frequently referred-to set of constraints (Felleman & Van Essen 1991, table 7), the number of optimal hierarchies found was even larger, with the algorithm yielding almost 200 000 visual hierarchies that all possessed a lower cost than solutions derived in previous manual analyses (Hilgetag *et al.* 1996, 2000b). The degeneracy of the solution space is caused by features of the systems' organization of the primate visual cortex (Hilgetag *et al.* 2000b), as well as by the use of a simple categorical cost function in traditional hierarchical analyses. Ultimately, approaches will have to be employed that use quantitative constraints for determining sequential connectivity patterns in the brain (Barone *et al.* 2000; Hilgetag *et al.* 2000b; Hilgetag & Grant 2001).

The solutions found by CANTOR also showed several organizational features that were not revealed by previous,

manually derived schemes of the same system, such as a larger number of hierarchical stages (between 12 and 18 levels) or a different composition of constraint violations. Using CANTOR's flexibility in designing cost functions, we also employed an alternative, more complex hierarchical cost, and weighted reliable and less reliable constraints differently (Hilgetag *et al.* 2000a). These settings did, however, not significantly alter the resulting hierarchical arrangements. Moreover, employing the CANTOR system in hierarchical analysis allowed us to computationally test hypotheses that pertain to the basis of the few remaining constraint violations in global visual hierarchies. The simulations suggested that one of the cortical visual areas, FST, might possess distinct subcomponents and they also predicted some of the subcomponents' connectivity with other cortical structures (Hilgetag *et al.* 2000b).

Hierarchical analysis of cortical connectivity provided a challenging test case for the stability and efficiency of CANTOR and the PUPS environment. The simulations for testing hypotheses about rule violations forced the system to process very large (>10 Mb) input object files in its persistent object space. Collecting up to 200 000 different solutions with an optimal hierarchical cost required CPU times of several months on the largest mainframe computer (a multi-processor DEC ALPHA) available to us. The computation was frequently interrupted by hardware and software failures, further stimulating the use of persistent mechanisms in PUPS, and, in particular, the incorporation of process state paging techniques in CANTOR.

(d) Further bioinformatic applications

The most complex bioinformatic system implemented under the PUPS environment to date is DAISY. DAISY is a CAT system that also has potential application in neuroinformatics, for instance in the automated screening of brain sections for specific classes of neurons. DAISY is a general-purpose pattern recognition tool that uses a set of nearest-neighbour classification engines (Lucas 1997; Alexander & Stonham 1979) that run in parallel within a computational cluster environment. It is capable of identifying arbitrary pattern vectors by comparing them with training sets of pattern vectors, the identities of which are known. For example, the system has been used both to screen and to identify visually similar species of Costa Rican ichneumonid wasps of the genera *Enicospilus* and *Pimpla* using imagery of the insects wings (Weeks *et al.* 1998, 1999; Gauld 1998).

From the standpoint of the PUPS environment, DAISY is, like ITG, an important test-bed application. As a biodiversity screening engine, for example, it has to be potentially able to screen thousands of species that span hundreds of genera in a timely manner via the Internet. The implication of this is that DAISY must be scalable, fault tolerant and inherently parallel; DAISY thus fully exercises the facilities of the PUPS environment. In order to function, it requires network-wide dynamic load balancing (to optimally use network resources), homeostasis (to make the system fault tolerant) and peer-to-peer process communication (to allow users to interact with the running system).

Experience with the DAISY system has led to a number of significant extensions to the PUPS environment that

will also be useful when building neuroinformatic applications of a similar level of size and complexity. These extensions include the following.

- (i) The use of sensitive input directories (SIDs) as repositories for input data: when data arrives in the directory, the application process associated with the directory is signalled and then takes appropriate action.
- (ii) The use of data-tagging techniques to facilitate the processing of data: data placed into SIDs that are associated with multiple applications may be tagged with a unique key. Only those processes that possess a lock that matches the key are allowed to process the data. This mechanism, which is based on a crude model of an immune system, permits systems of co-operating processes of arbitrary complexity to be constructed, the global topological properties of which are emergent. This means the implementor only need define a set of input keys that processes require to read input data from a SID, and a set of output keys that processes use to stamp data they place in the SID. The keys then implicitly define the spatial and temporal topology of the process network.

5. DISCUSSION

(a) *Persistent computations*

There is a single theme that unifies all of the quite disparate applications that have been implemented using the PUPS environment. In order to complete their payload computations successfully, applications such as DAISY, CANTOR and even the ITG stereo system all have long lifetimes compared with typical computer programs. In the case of DAISY, the lifetime of the running system is liable to be measured in years. Being long-lived gives rise to its own unique set of problems. Events that occur with a low probability in the case of applications the lifetimes of which are measured in hours, will be almost certainly occur in applications that run for weeks, months or years. Thus, persistent computations must expect and be able to deal with problems such as crashing host computers, software failures and deletion of or damage to an application's data or its address space, as well as acute fluctuations of important system resources, such as disk space, CPU-time and memory.

This means that persistent applications and the environments that support them must embrace homeostatic mechanisms at many levels. Consequently, PUPS freely appropriates many biological mechanisms including:

- (i) non-centralized control of computations;
- (ii) use of cumulative evolution in order to control the emergent properties of computations;
- (iii) provision for self-assembling applications (the properties of which are emergent);
- (iv) redundancy (applications must fail gracefully);
- (v) irritability—applications continuously monitor their execution environment and can therefore react to changes in it (e.g. migrating off a malfunctioning processor, waiting for appropriate resources to be released when other applications crash, reacting to input data placed in a SID, etc.);
- (vi) intelligence—persistent systems must be able to learn about their environment and then use that knowledge to enhance their chance of survival.

The function of the PUPS system, as we have discussed at some length, is to provide an environment that facilitates the implementation of software for running persistent computations. In order to do this effectively, both the PUPS environment and the applications that are built on it have behaviours that are superficially similar to those of primitive biological systems. This should not be surprising, as biological objects are effectively a naturally occurring class of persistent computation.

(b) *Alternative approaches to the analysis of complex data*

(i) *Databases and storage methods*

Flattened databases—e.g. traditional relational database management systems (RDBMSs) such as Ingres, Mariposa, or Oracle—fold relationships, which may be best expressed in a space of high dimensionality, into tables that contain both the database objects and the relationships between them. Although this approach works well for many commercial applications with a small number of relationships between objects, it fails to deal efficiently with the kind of complexity inherent in bio- and neuroinformatic applications, for several reasons. There may be significant time penalties incurred as a consequence of the flattening inherent in this sort of approach. In many RDBMSs, there is little or no provision for metadata, for example executable programs or dynamically linkable objects. Optimization, for example the stochastic search methods described here, may be difficult to implement given the underlying software structure of the RDBMS.

The PUPS environment overcomes these difficulties in three ways.

- (i) Support for persistent object stores means that complex databases may be loaded directly into PUPS applications without any need for flattening.
- (ii) The internal model of data within PUPS (i.e. the CANTOR standard) is a generic object that may represent data, metadata or a pointer (signifying a relationship) to another object. This model is very flexible. For example, data may be ASCII text, imagery, executable programs, or dynamically linkable functions. The pointers used by the PUPS system are doubly indirected, which means that in a CANTOR database, node content is totally independent of database topology.
- (iii) The internal data schemas used by CANTOR are compatible with the emergent extensible mark-up language (XML) standard, which is being increasingly used in bio- and neuroinformatic databases. This means that the XML standard may be employed by PUPS applications for storing data in a portable fashion, which will permit interoperability with other bio- and neuroinformatic database and data-processing systems.

(ii) *Optimization algorithms*

Traditionally, methods such as non-metric multi-dimensional scaling, artificial neural networks (ANNs),

principal component analysis (PCA) and linear discriminant analysis have been used to elucidate optimal properties of the kinds of fuzzy datasets inherent in bio- and neuroinformatics. See Hilgetag *et al.* (2001) for a more detailed review of analytical methods employed in the investigation of relational datasets. There are a number of problems with these approaches. Many of these methods assume that datasets are essentially unimodal. As a consequence of this, these methods fail to extract global properties of the data. With the exception of ANNs, almost all of the preferred methods of analysis also assume that the data space does not deviate very far from linear order. For the kind of analyses described here, and a whole host of other bio- and neuroinformatic optimizations, this is clearly not the case and application of linear methods to nonlinear data will almost certainly lead to misleading results. Additionally, some of these methods, for example ANNs and PCA, do not scale well. Addition of new data to the system means that the entire neural network (for ANNs) or set of eigenvectors (for PCA) may need to be recomputed. Implicitly, all these standard analysis methods assume closed datasets. In some bio- and neuroinformatic applications, for example the elucidation of neuronal morphologies from microscopically examined brain slices, it is clear that the variation of any training set of neurons of a given morphology type will not span the global variation of that type. Thus, methods such as ANNs that employ back-propagation will tend to overtrain, which may lead to erroneous results.

The PUPS system attempts to overcome the difficulties described above by providing optimization algorithms, such as stochastic nearest-neighbour classifiers. Such algorithms have the advantages of being relatively fast, amenable to nonlinear data and readily scalable. If necessary, they can be parallelized. Naturally, the stochastic optimization techniques implemented in the PUPS libraries and

used by applications such as CANTOR are not a panacea. Although they are better at finding global optima than linear methods, and do not suffer from the scaling or overtraining problems inherent in ANNs and PCA, methods using stochastic optimization may still fail to reach a global minimum. (This means that we cannot be sure that we have found the 'best' hierarchy or cluster in the case of CANTOR or the 'right' taxon in the case of DAISY.) This is a consequence of the way in which stochastic optimizers search and sample large spaces. In order to improve the convergence characteristics of the algorithm, the assumption is made that the global minimum is relatively close to the current best estimate of the global minimum. An appropriate proximity between these two values depends on the parameters set by the experimenter. Failure to ascribe realistic values will lead to a system that fails to converge, i.e. does not settle to a stable solution, or that converges to a local minimum. The correct parameterization of global optimization problems, therefore, often requires significant insight on the part of the experimenter. Although there are optimization techniques that are guaranteed to find global minima, these brute-force methods are extremely time consuming, and even for moderately large search spaces, convergence will not be achieved in a finite time on conventional Von Neumann computer architectures. It is, however, theoretically possible to efficiently solve this kind of optimization using quantum computation. For example, the space of pattern vectors that is searched by the DAISY algorithm is amenable to Grover's algorithm (Grover 1996), given appropriate hardware developments.

This work has been supported by the Wellcome Trust and the UK Government Darwin Initiative. Thanks to Gully Burns for help with figure 1.

APPENDIX A

log	< on off >	switch server transaction logging < on > or < off > . If transaction logging is enabled the server logs all transactions from connected PSRP servers and notes whether errors occurred in processing the transactions to <i>stdout</i> .
app_verbose	< on off >	switch server error logging < on > or < off > . If error logging is enabled the server logs errors (e.g. failure to allocate resources etc) to log file associated with <i>stdout</i> .
show		display user defined transaction processing functions (dispatch functions) for current (connected) server.
clients		display clients connected to current server. PUPS/PSRP currently supports up to 256 simultaneous PSRP sessions per server (e.g. up to 256 users or user-processes may simultaneously talk to a given server).
bindtype		display current object binding (static or dynamic). If the object binding is dynamic additional transaction processing functions may be attached to a running server via a PSRP session. Static binding implies that PSRP sessions are <i>status only</i> .
help		display online help information.
atexit		display list of (PUPS) application exit functions. Functions in this list are executed when the server process exits and are typically used to free dynamic resources and remove temporary files.
atckpt		display list of functions which will be executed by the current server process when it is checkpointed. Checkpoint functions are typically used to detach dynamic functions and shared heaps prior to a process producing a checkpoint file.
atrestart		display list of functions that are invoked when server process is restarted using a checkpoint file. Restart functions typically re-attach the dynamic objects that were previously detached by the <i>atckpt</i> directive.
auto_ckpt	[on off]	invoke homeostat which automatically checkpoints server process [on], disable automatic checkpointing [off] or get automatic checkpoint status (no arguments).
authenticate		tells server process to authenticate privileged operations invoked via the PSRP interface by demanding an authentication token. This facility permits users to provide secure access to PSRP server processes that they own.
*unprotect		disable authentication (for privileged operations).
*migratory		enable MOSIX process migration. Migratory processes make use of the MOSIX resource management system, transparently migrating from node to node within a process cluster in order to make the best use of the computational resources available.
*sedentary		disable MOSIX process migration. Process cannot migrate and must remain on the node on which it was created.
pstat		display/proc filesystem status entry. This is an interface to the LINUX <i>pstat</i> command.
lstat		display <i>link file locks</i> currently held by the server process. Link file locking is a PUPS mechanism which extends the LINUX/UNIX <i>lockf</i> network-wide file locking mechanism, and allows processes to access shared file system objects in an orderly manner.
vitstat		display status of PUPS <i>virtual interval timers</i> . PUPS/PSRP server processes use virtual interface timers that regularly interrupt processing of the <i>computational payload</i> in order to service both built-in and user defined <i>homeostats</i> .
sigstat		display non-default signal handlers, that is signal handler which have a service function associated with the (e.g. <i>not SIG_IGN</i> or <i>SIG_DFL</i>).
sigstat	< sig list >	display detailed data for selected signal handlers. For the given signals the <i>handler function</i> and the signals which are <i>blocked</i> while the handler executes are displayed.
maskstat		display signal mask (and pending signals) for server process.
dstat		display open file descriptors. A list of open file descriptors associated with the PSRP server are displayed together with the names of filesystem objects with which they are associated. In addition, if a given descriptor is homeostatically protected, the name of the homeostat function is displayed.
cstat		display child processes associated with current PSRP server (e.g. processes created by the current PSRP server using the UNIX/LINUX <i>fork</i> system call).
sicstat		display open slaved interaction client channels. The slave channel is used by a PSRP server to talk to a (slaved) PSRP client process which it may use to initiate <i>peer-to-peer transactions</i> with other PSRP server processes.
ostat		display orifices (portals exporting dynamic objects) associated with current PSRP server.
cronstat		display list of actions which are to be scheduled by this PSRP server at user specified times (this is a per-process version of UNIX/LINUX <i>cron</i> utility).
*schedule	< f > < t > < func >	schedule function < func > to run between times < f > ant < t > . < func > may be any kind of transaction (dispatch) function, etc.
*unschedule	< index >	unschedule a cron function. < index > is the slot index returned by the <i>cronstat</i> built-in.

continued

APPENDIX A *continued*

rusage		PSRP interface to the UNIX/LINUX <i>rusage</i> system call. Displays current resource usage (memory, open files, CPU) for the server process.
rset		PSRP interface to the UNIX/LINUX <i>rset</i> system call. Permits PSRP client to set limits for server memory and CPU usage, etc.
showaliases	< n >	show aliases of PSRP transaction (dispatch) function < n > . Aliases are additional names which are associated with a given PSRP transaction function.
alias	< n > < a >	alias < n > to < a > . This mechanism permits a user to customize the name of a given service. After aliasing < n > to < a > the names < n > and < a > are synonymous and both refer to the same underlying service.
unalias	< a >	remove alias < a > . < a > is no longer a synonym on < n > .
*vitab	[< n > shrink]	display [or set to < n > or Procrustes shrink] number of virtual timer table slots. This sets the upper limit for the number of virtual timers the server process may have active. The <i>shrink</i> option sets the upper limit to the number of virtual timers currently active.
*ctab	[< n > shrink]	display [or set to < n > or Procrustes shrink] number of child table slots. This sets the upper limit for the number of child processes a given server can create. The <i>shrink</i> option sets the upper limit to the number of children currently active.
*ftab	[< n > shrink]	display [or set to < n > or Procrustes shrink] number of file table slots. This sets the upper limit for the number of file descriptors a given server can have open <i>simultaneously</i> . The <i>shrink</i> option sets the upper limit to the number of files currently open.
*htab	[< n > shrink]	display [or set to < n > or Procrustes shrink] number of shared heap table slots. This sets the upper limit for the number of shared heaps which the server process is allowed to have attached <i>simultaneously</i> . The <i>shrink</i> option sets the upper limit to the number of shared heaps currently attached to the server process.
*otab	[< n > shrink]	display [or set to < n > or Procrustes shrink] number of orifice table slots. This sets the upper limit for the number of dynamic objects which the current server process can simultaneously bind. The <i>shrink</i> option sets the upper limit to the current number of bound dynamic functions.
*new	< n > < h > < term >	create instance (< n >) of server on host < h > . PSRP interface to LINUX/UNIX <i>fork</i> system call: duplicates process.
*overlay	< command >	overlay server process with < command > . PSRP interface to LINUX/UNIX <i>execv</i> system call.
*overfork	< command >	over-fork server process with < command > . Causes server process to suspend execution and to <i>fork</i> a child process which is used to <i>exec</i> . The over-fork facility permits the user to dynamically insert foreground monitoring tools into a running PUPS processes pipeline (e.g. X based visualization tools). When the tool exits, the <i>overforked server process</i> (typically xcat) resumes execution.
*autosave	< on off >	switch automatic saving of dispatch table (at exit) on or off. If autosave is enabled a file is written which enables the server process to reload all dynamic functions and/or aliases currently associated with it.
autosave		display status of automatic dispatch table saving (at exit)
*save	< n >	save dispatch table (to PSRP resource file < n >)
*load	< n >	load PSRP resource file < n > (appending objects to current dispatch table)
*reset		reset dispatch table (returning it to its default state). All dynamic functions currently bound to server are detached and all aliases to (static) transaction (dispatch) functions are removed.
*live	< f1 > , < f2 > , ...	protects the list of specified files against deletion using the PUPS <i>default file homeostat</i> . With no options lists homeostatically protected files.
*dead	< f1 > , < f2 > , ...	unprotects list of specified files (if any file in the list is protected its protection will be revoked). With no options lists open files which are not currently homeostatically protected.
*detach	< item-list >	detach PSRP objects function in < item-list > . If itemlist is 'all' detach <i>all</i> dynamic transaction (dispatch) functions.
*terminate		terminate (this) PSRP server process. PSRP interface to the LINUX/UNIX <i>exit</i> system call.
*dll	< name > < objectDLL >	bind dynamic function < name > in dynamic link library [DLL] to the current server. After the binding is done, the bound function appears as a dynamic transaction (dispatch) function with (root) name < name > .
hostat	< ol > ... < on >	show statistics for tracked heap object < tracked object > . PUPS/PSRP provides a (debugging) mechanism which enables heap objects (e.g. objects allocated via the <i>malloc</i> package) to be viewed. This mechanism is useful in detecting and correcting memory allocation errors.
hostat		show statistics for all tracked heap objects

Table A1. Built-in PSRP server functions.

(‘*’ implies that a given directive is privileged and that authentication may be required before it is executed.)

```

/*-----
Purpose: Skeleton PUPS/PSRP application
Authors: M.A. O'Neill
Digital Vision
Didcot
Oxfordshire
OX11 8QY

Dated: 28th March 2001
Version: 1.01
E-mail: mao@sable.ox.ac.uk
-----*/

/* These header files MUST be included. They are located in the include.libs */
/* directory of the standard pups distribution. */
#include < me.h >
#include < utils.h >
#include < netlib.h >
#include < psrp.h >
#include < hiplhdr.h >
#include < unistd.h >
#include < string.h >
#include < errno.h >
/*-----

Get application information for slot manager . This application information is returned when the user specifies either the
\usage -help or \version command tail flags ...
-----*/

/* Slot information function */
_PRIVATE void $APPNAME(int level)
{ fprintf(stderr, "int dynamic [PUPS/PSRP] application $APPNAME 1.10: [ANSI C]\n");

if(level > 1)
{ fprintf(stderr, "(c) $DATE Digital Vision, Didcot\n");
fprintf(stderr, "Author: M.A. O'Neill\n");

/* Description of this PUPS/PSRP application in this print statement. More print statements can be */
/* added here if required */
fprintf(stderr, "$APPDES (PUPS format)\n\n");
}
else
fprintf(stderr, "\n");

fflush(stderr);
}

/* Application usage function */
_PRIVATE void $APPNAME.usage()

{ /* Put your usage information here—see for example embryo.c */
/* to look at the format for this */

/* Standard blurb about the signals required to support */
/* CKPT and PSRP protocols */
/* NOTE: unless we are running on an appropriately */
/* modified LINUX kernel the signals defined below are */
/* synonyms for standard POSIX signals. These synonyms are */
/* defined in utils.h and psrp.h */
fprintf(stderr, "\n\nSignals\n\n");

#ifdef PSRPENABLED
/* Signals which are used by PUPS/PSRP if the application has an active PSRP interface */
fprintf(stderr, "SIGINIT SIGCHAN SIGPSRP: Process status [PSRP] request (protocol%2.2f)\n", PSRP_PROTO-
COL_VERSION);
fprintf(stderr, "SICLIENT: tell client server is about to segment\n");

```

(Cont.)

Table A1. (Cont.)

```

#ifdef THREAD/SUPPORT

/* Signals which are active if the application expects to run more than one (real) thread */
/* of control (using the pthread [POSIX] thread API */
fprintf(stderr, "SIGTHREAD: tell client its service thread (on PSRP server) has been terminated\n");
#endif

/* This signal is simply ignored. It is sent by other processes which wish to find out whether this */
/* server is alive. If the server is alive the process which sent the SIGALIVE signal get s a non */
/* zero return (to its kill() or pupskill() call. If dead this call will return (-1) */
fprintf(stderr, "SIGALIVE: check for existence of client on signal dispatch host\n");

#ifdef CKPT
/* Signals which are used when applications supports (Tennessee) checkpointing */
if(checkpointing == TRUE)
fprintf(stderr, "SIGCHECK SIGVACATE: CKPT job control\n");
#endif
#endif

fprintf(stderr, "\n");
fflush(stderr);
}

#ifdef SLOT
#include < slotman.h >
/EXTERN void (* SLOT)() = $APPNAME_slot;
/EXTERN void (* USE)() = $APPNAME_usage;
#endif

/* -----
Public variables and function pointers used by this application. These are variables which are global to this
application .and can be accessed by any function within it . . .
-----*/

/* -----
Functions which are private to this application . These functions are private to this .c file (e.g. static functions) . Note use of /
PRIVATE rather than static is encouraged as it enhances the readability of the code . . .
-----*/

/* -----
Variables which are private to this module. These variables are private to this .c application (e.g. static variables)
-----*/

/* -----
Software I.D. tag (used if CKPT support enabled to discard stale dynamic
checkpoint files) . Also used by the built-in\version command tail flog to abstract build information . . .
-----*/

#define VTAG 1
extern int app_vtag = VTAG;

#ifdef CKPT_SUPPORT
extern int checkpointing;
extern int condor_restarts;
#endif

/* -----
Main function. Note that this is pups/main() rather than main(). Adherence to the pups/main() convention means that
checkpointable versions of the application may be built transparently by simply changing a compilation flag . . .
-----*/

```

(Cont.)

Table A1 (*Cont.*)

```

_PUBLIC int pups_main(int argc, char *argv[])

{

#ifdef PSRP_ENABLED
/* Ignore any clients who may be trying to attach to us until we are initialized */
/* failure to do so could result in deadlock */
psrp_ignore_requests();
#endif

/*-----
Get standard items form the command tail. . .
-----*/

std_init(TRUE, /* TRUE if code GNU copyleft, FALSE otherwise */
&argc,
"1.00",
$AUTHOR, /* Substitute the name of the program author */
"$APPNAME", /* Substitute name of this application */
"$DATE", /* Substitute version date */
argv);

/* Application specific command tail arguments should go here */
/* embryo.c has some examples of how it should be done */

/* Complain about any unparsed arguments . If we do have any */
/* unparsed arguments t_arg_errs() will call pups_exit() */
t_arg_errs(argc,argv,args);

#ifdef PSRP_ENABLED
/* Initialize the PSRP system here using psrp_init() */
/* Note any PSRP dynamic load option you do not want */
/* is simply excluded from the first set of OR'd args */
/* If you don't want homeostasis protection for the */
/* server messaging channels omit PSRP_HOMEOSTATIC_STREAMS */
psrp_init(PSRP_STATIC_FUNCTION 1 PSRP_STATIC_DATABAG|PSRP_HOMEOSTATIC_STREAMS,NULL

/* Load default dispatch table for this applications. Loads */
/* dynamic objects which have been previously attached to this */
/* application and remembers any aliases we may have had for */
/* dispatch table objects */
(void)psrp_load_default_dispatch_table(stderr);

/* Once we have initialised the PSRP system, attached any dynamic objects */
/* and loaded any alias tables lets tell the world we are open for business */
psrp_accept_requests();
#endif

/* Any payload functions which attach or create objects like files and shared heaps */
/* should register an exit function here, so that any temporary objects are destroyed */
/* by pups_exit() */

/*-----
Static payload of application here - dynamic payloads are loaded using the PSRP
bind functions ...
-----*/

/* Exit nicely clearing up any mess we have made */
pups_exit(0);
}

```

Table A2. Example code skeleton for a generic PUPS/PSRP application.

(Further examples of PUPS/PSRP code fragments, including a skeleton DLL and code to attach and use shared heaps, are available at the PUPS Web site: <http://www.pups.org.uk>.)

clean:	shell (<i>cs</i> <i>h</i>) script to clean distribution directories.
config:	configuration directory. Holds rules that automatically generate makefiles for different types of software architecture (e.g. UNIX, LINUX, OSF-1, etc.) and PUPS system type (e.g. vanilla, shared memory, cluster, etc.).
examples:	example PUPS application sources (written in ANSI-C).
include.libs:	PUPS/PSRP header (.h) file directory.
pupsconf:	automatic configurator script (generates makefiles for a given architecture and PUPS system type).
pupsuname:	shell script that returns PUPS system type e.g. Ix86.linux for PUPS built under a LINUX kernel running on Intel 80x86 CPU.
pups-webdist:	shell (<i>cs</i> <i>h</i>) script that automatically downloads and compiles complete PUPS distribution from a distribution website.
pups_bootstrap:	shell (<i>cs</i> <i>h</i>) script to automatically bootstrap tools that are required for <i>pups_webdist</i> and the PUPS JIT (just in time) compilation tools to function.

Table A3. Contents of standard PUPS/PSRP distribution filesystem (*dist.supups*).

REFERENCES

- Alexander, I. & Stonham, T. 1979 Guide to pattern recognition using random access memories. *IEEE Proc. Comp. Digital Techniques* **2**, 29–40.
- Barak, A., Geday, S. & Wheeler, R. G. 1993 The MOSIX distributed operating system: load balancing for UNIX. In *Lecture Notes in Computer Science*, p. 672. Springer.
- Barone, P., Batardiere, A., Knoblauch, K. & Kennedy, H. 2000 Laminar distribution of neurons in extrastriate areas projecting to visual areas V1 and V4 correlates with the hierarchical rank and indicates the operation of a distance rule. *J. Neurosci.* **20**, 3263–3281.
- Bridge, P. D., O'Neill, M. A. & Fatehi, J. 2001 Stochastic annealing for determining species boundaries from variable DNA sequences. *Taxon*. (Submitted.)
- Burns, G. A. P. C. & Hilgetag, C. C. 2000 The computational representation and analysis of neuroanatomical knowledge: limiting the problem of information overload in neuroscience. *FASEB J.* **14**, A544.
- Denos, M. I. & O'Neill, M. A. 1992 A practical approach to the stereomatching of urban imagery. *Image Vis. Comp.* **10**, 89–98.
- Denos, M. I. & O'Neill, M. A. 1996 ITG: a system for the automatic generation of terrain maps from optical stereo imagery. *Image Vis. Comp.* **14**, 225–236.
- Felleman, D. J. & Van Essen, D. C. 1991 Distributed hierarchical processing in the primate cerebral cortex. *Cereb. Cortex* **1**, 1–47.
- Gauld, I. D. 1998 A survey of the Ophioninae (Hymenoptera: Ichneumonidae) of tropical Mesoamerica with special reference to the fauna of Costa Rica. *Bull. Br. Mus. (Nat. Hist.), Entomol. ser.*, vol. 57.
- Grover, L. K. 1996 A fast quantum mechanical algorithm for database search. *ACM Symp. Theor. Comp.*, pp. 212–219. Philadelphia.
- Gruen, A. W. 1985 Adaptive least squares correlation: a powerful image matching technique. *S. Afr. J. Photogrammetry, Remote Sensing & Cartography* **14**, 175–187.
- Hilgetag, C. C. & Grant, S. 2001 Uniformity and specificity of long-range corticocortical connections in the visual cortex of the cat. *NeuroComputing*. (In the press.)
- Hilgetag, C. C., O'Neill, M. A. & Young, M. P. 1996 Indeterminacy of the visual system. *Science* **271**, 776–777.
- Hilgetag, C. C., Burns, G. A. P. C., O'Neill, M. A., Scannell, J. W. & Young, M. P. 2000a Anatomical connectivity defines the organization of clusters of cortical areas in the macaque monkey and the cat. *Phil. Trans. R. Soc. Lond. B* **355**, 91–110.
- Hilgetag, C. C., O'Neill, M. A. & Young, M. P. 2000b Hierarchical organization of macaque and cat cortical sensory systems explored with a novel network processor. *Phil. Trans. R. Soc. Lond. B* **355**, 71–89.
- Hilgetag, C. C., Kötter, R., Stephan, K. E. & Sporns, O. 2001 Computational methods for the analysis of brain connectivity. In *Computational neuroanatomy* (ed. G. Ascoli). Totowa, NJ: Humana. (In the press.)
- Jouve, B., Rosenstiehl, P. & Imbert, M. 1998 A mathematical approach to the connectivity between the cortical visual areas of the macaque monkey. *Cereb. Cortex* **8**, 28–39.
- Knuth, D. 1981 *Seminumerical algorithms*. In *The art of computer programming*, 2nd edn, vol. 2, pp. 116ff. Reading, MA: Addison-Wesley.
- Litzkow, M., Livny, M. & Mutka, M. W. 1988 Condor—a hunter of idle workstations. *Proc. 8th Int. Conf. Distrib. Comp. Syst.* 104–111.
- Lucas, S. M. 1997 Face recognition with the continuous n-tuple classifier. *Proc. Br. Mach. Vis. Conf.* 222–231.
- Maunsell, J. H. & Van Essen, D. C. 1983 The connections of the middle temporal visual area (MT) and their relationship to a cortical hierarchy in the macaque monkey. *J. Neurosci.* **3**, 2563–2586.
- Milner, A. D. & Goodale, M. A. 1996 *The visual brain in action*. Oxford University Press.
- Otto, G. P. & Chau, T. K. W. 1989 Region growing algorithm for the matching of terrain imagery. *Image Vis. Comp.* **7**, 83–93.
- Peters, A. & Rockland, K. S. (eds) 1994 Primary visual cortex in primates. In *Cerebral Cortex*, vol. 10. New York and London: Plenum.
- Plank, J. S., Beck, M. & Kingsley, G. 1995 Libckpt: transparent checkpointing under UNIX. *Proc. USENIX Winter Tech. Conf. New Orleans Louisiana, 16–20 January 1995*.

- Scannell, J. W., Burns, G. A. P. C., Hilgetag, C. C., O'Neill, M. A. & Young, M. P. 1999 The connectional organization of the cortico-thalamic system of the cat. *Cereb. Cortex* **9**, 277–299.
- Sporns, O., Tononi, G. & Edelman, G. M. 2000 Theoretical neuroanatomy: relating anatomical and functional connectivity in graphs and cortical connection matrices. *Cereb. Cortex* **10**, 127–141.
- Stephan, K. E., Hilgetag, C. C., Burns, G. A. P. C., O'Neill, M. A., Young, M. P. & Kötter, R. 2000 Computational analysis of functional connectivity between areas of primate cerebral cortex. *Phil. Trans. R. Soc. Lond. B* **355**, 111–126.
- Ungerleider, L. G. & Mishkin, M. 1982 Two cortical visual systems. In *Analysis of visual behaviour* (ed. D. G. Ingle, M. A. Goodale & R. J. Q. Mansfield), pp. 549–586. Cambridge, MA: MIT Press.
- Van Essen, D. C., Anderson, C. H. & Felleman, D. J. 1992 Information processing in the primate visual system: an integrated systems perspective. *Science* **255**, 419–423.
- Weeks, P. J. D., O'Neill, M. A., Gauld, I. D. & Gaston, K. J. 1998 Automating the identification of insects: a new solution to an old problem. *Bull. Entomol. Res.* **87**, 203–211.
- Weeks, P. J. D., O'Neill, M. A., Gaston, K. J. & Gauld, I. D. 1999 Automating insect identification of insects: exploring the limitations of a prototype system, *J. Appl. Ent.* **123**, 1–8.
- Willmott, A. P., Ellington, C. P. & Thomas, A. L. R. 2001 Flow visualisation and unsteady aerodynamics in the flight of the hawkmoth *Manduca sexta*. *Phil. Trans. R. Soc. Lond. B*. (In the press.)
- Young, M. P. 1992 Objective analysis of the topological organization of the primate cortical visual system. *Nature* **358**, 152–155.
- Young, M. P., Scannell, J. W., O'Neill, M. A., Hilgetag, C. C., Burns, G. & Blakemore, C. B. 1995 Non-metric multidimensional scaling in the analysis of neuroanatomical connection data from the primate visual system. *Phil. Trans. R. Soc. Lond. B* **348**, 281–308.