Chapter 21
Large-Scale Simulation Using Parallel GENESIS

NIGEL H. GODDARD AND GREG HOOD

21.1 Introduction

PGENESIS is a parallel form of GENESIS that enables simulation of very large models. Simulation models are critical for integration of behavioral data with anatomical and physiological data. Although explanations of behavioral data are possible without resort to neural simulation models (Chomsky 1957, e.g.), those integrative accounts that make contact with the anatomical and physiological data require large-scale simulation models at the neural level. The scale of the models required can be seen in theories about the function of the hippocampus in learning and memory (McClelland and Goddard 1996, Levy 1996). These theories assert that statistical properties of firing rates, synaptic transmission efficiencies, and connection structures are crucial in explaining information processing in the hippocampus. The validity of these statistical properties is conditioned on sufficient sample sizes that cannot hold if the model scales down the real system by more than one or two orders of magnitude. Even scaling down by two orders of magnitude leaves us with very large models that, as we shall see, go beyond the capabilities of existing simulation environments.

Two developing data acquisition methodologies are driving this interest in larger scale models. Technologies for imaging the nervous system, particularly functional magnetic resonance imaging (Belliveau, McKinstry, Buchbinder, Weisskoff, Cohen, Vevea, Brady and Rosen 1991), allow us for the first time to test hypotheses embodied in systems level models and to relate these models to behavior. Multi-electrode recording devices (Wilson, McNaughton and Stengel 1992), which are now migrating from the small number of originating labs to a larger group of users in diverse labs, are already delivering individual spike data for over one hundred cells simultaneously, allowing modelers to refine hypotheses about information representations employed in particular systems.

Effective use of a parallel computer for simulation requires that the problem be partitioned in such a way as to minimize the overhead associated with communication and synchronization between processors, while at the same time balancing the amount of work done by each processor. Overhead is minimized by reducing the communication between components of the simulation that reside on different processors, and by maximizing the the extent to which the components of the simulation on different processors can proceed without synchronizing. There are two characteristics of communication hardware (e.g., Ethernet) that are referred to in this chapter. *Bandwidth* is the effective rate at which data can be sent over the medium. *Latency* is the time between the transmission of data by one processor and their reception by another. Two classes of neural simulations can often exhibit low overhead:

1. **Parameter search**. Global optimization algorithms for fitting a parameterized model to data require that many parameterizations be evaluated. In neural models evaluation of a particular parameterization is often best achieved by running the model. There are many optimization algorithms that can be parallelized so that many parameterizations are evaluated simultaneously, including parallel genetic algorithms (Collins and Jefferson 1991) and parallel simulated annealing (Azencott 1992). Each parameterization of the model can be run on a separate processor. The communication costs are small: the parameters must be transferred on startup, and the fitness value returned at the end of the evaluation run of the model. Synchronization costs can be very low because models are run independently. The amount of synchronization required depends on the particular search strategy, as discussed further in Sec. 21.7.

2. **Network models**. Network models often exhibit two characteristics that closely match the underlying hardware of parallel platforms, if we assume a partition of the model that keeps the compartments of each neuron on a single processor so that the communication is exclusively in the form of spikes. First, spike rates are low compared with the time step for integration within the cell, thus communication bandwidth is low as few spikes need to be communicated on each time step. Second, axonal delays are typically one or two orders of magnitude greater than the time step, so that cells need not be simulated in lock step. Simulation time on different processors can differ as long as every spike is delivered to its destination within the axonal delay period. These characteristics match those in parallel platforms, in which bandwidth is often limited and latency is often high.

In contrast, distributing a simulation of an electrotonically connected model over many processors will incur much larger overhead. This is because at each time step the processors must exchange data values and, implicitly, synchronize. There is extensive communication, and processors must wait for all to finish a time step before they can proceed to the next step. In this chapter will introduce the use of PGENESIS for the two classes of models enumerated above: parameter search and network models. We also discuss hybrid simulations in which parameter search is performed on a network model that itself runs in parallel.

21.2 Classes of Parallel Platforms

The three major classes of parallel platforms are (1) networks of workstations (NOWs), (2) symmetric multiprocessors (SMPs) and (3) non-uniform shared memory massively parallel processors (NUMA MPPs). In addition there are hybrid versions of NOW and MPP in which each node of the machine is itself an SMP. The two most important efficiency questions when deciding which platform to concentrate on are: how well its communication characteristics match those needed by the simulation task; and how the architecture scales up to the number of nodes that the model can use.

As stated before, the important communication characteristics are bandwidth and latency, and of these latency is usually more critical for neural models. Low latency and high bandwidth is the goal. To simplify vastly, NOWs typically have relatively high latency and low bandwidth. Thus they are suitable for coarse-grained parallel tasks such as a parameter search task where evaluation of a single individual takes an appreciable amount of time. They may also be suitable for network models with very long lookahead that may not be adversely affected by high latency. Other factors to consider with a NOW platform are whether one has exclusive access to the processors and how much disk traffic the simulation generates. Without exclusive access, it is very hard to partition a network model efficiently. NOW platforms are typically not set up well for massive transfers to and from shared disk for many processors.

SMP platforms (e.g., Origin, Convex, Ultrasparc) have very low latency and high bandwidth for small to medium numbers of processors (e.g., up to 16), but these characteristics do not scale well to a high degree of parallelism because the underlying hardware communication medium — a shared bus — does not scale. Nevertheless they can be very effective for development of small to medium scale PGENESIS models and are often easier to use than MPPs or NOWs.

NUMA MPP machines (e.g., Cray T3E) are the ideal platform for the largest, most highly parallel PGENESIS tasks. The latency and bandwidth characteristics are vastly better than NOW and approach SMP, and the architecture can scale into the hundreds of processors on parameter search tasks, and into the tens of processors for well-distributed network models. These machines are at the high end of high performance computing and so are designed to balance processor speed with memory latency and bandwidth and disk latency and bandwidth. The largest network models will almost certainly require them. However, the NOW or SMP platforms may be more cost-effective for large-scale parameter search tasks.

21.3 Parallel Script Development

Script development for PGENESIS is an exercise in parallel programming where many processes are running simultaneously. Each process (called a *node*) is running one instance of a single parallel script. For modelers familiar with serial programming, including all GENESIS users, the transition to parallel programming requires some conceptual leaps. For example, one can no longer assume that because statement B in a script comes after statement A that B will be executed after A. The order depends on which nodes the statements are executed by, and what synchronization events (e.g., barriers) occur between them.

We have found that PGENESIS simulations are best developed in the following order.

1. For all models, develop and debug the single cell prototypes using serial GENESIS.
2. (a) For network models, decide how the network should be partitioned, i.e., which GENESIS elements should go on which nodes. It is best to implement the scripts in a scalable fashion so that the number of nodes can be varied by changing a run-time parameter.

   (b) For parameter search tasks, develop the scripts to run and evaluate a single individual, and the scripts that control the optimization. The optimization scripts should be parameterized so that they will run with any number of nodes.
3. First try out your scripts on a single processor — a desktop workstation is often most convenient for this stage. Make sure that your scripts run correctly using the minimum number of nodes (a single node, if possible). Also, be sure the scripts will run in the background, without XODUS or any interactive input.
4. Continue to use the single processor platform, but increase the number of nodes in the simulation. This will show up errors related to the partitioning of the problem across more than one node.
5. Run the scripts on a multiprocessor platform with a small number of processors. For example, a small symmetric multiprocessor or a small number of networked workstations. This will show up errors due to assumptions about execution order.
6. Run the full scale simulations on the largest machine you need and have access to. By this time your scripts should be well-debugged. Typically debugging is difficult on

the largest platforms, so it is prudent to do as much debugging as possible on smaller machines.

21.4 Script Language Programming Model

A PGENESIS simulation consists of a set of independent processes (*nodes*) that can communicate via script language commands and can cooperate in a simulation via GENESIS messages between elements residing on different nodes. Serial GENESIS forms the core of each of these processes. As in serial GENESIS, execution of a simulation including setup and stepping, is controlled by scripts. Use of XODUS is discussed in Sec. 21.8. However, script programming for PGENESIS introduces additional complexities. It is critical that a user of PGENESIS review and understand these issues before attempting to run PGENESIS.

21.4.1 Parallel Virtual Machine

PGENESIS is built on top of the Parallel Virtual Machine (PVM) software system (Geist, Beguelin, Dongarra, Jiang, Manchek and Sunderam 1994), which provides the illusion of a parallel platform. The PVM system may run on a single CPU or multiple CPUs, possibly of different types. In the rest of this chapter, when we refer to the “parallel platform” we mean the illusion provided by PVM. When we refer to the “parallel machine,” we mean the physical set of CPUs and the network connecting them on which PVM is running. An executing PVM program consists of user processes, typically one per CPU, which communicate via the PVM daemon that runs on each participating CPU. In PGENESIS, each user process is an independent GENESIS simulation, which we call a *node* of the parallel simulation. Nodes are uniquely identified by a node number (consecutive integers starting at zero). These nodes may be grouped into *zones* — nodes within a zone have their simulation time kept more or less synchronized, whereas simulations in different zones may run relatively independently. Thus, a parameter search algorithm would typically run many simulations independently with each node in a separate zone, whereas a large network model would typically run with all nodes in a single zone.

21.4.2 Namespace

PGENESIS currently provides a private-namespace programming model. This means that each node has no knowledge of the elements that reside on other nodes. This implies that every reference to an element on another node must specify the node explicitly. It is envisioned that a shared-namespace programming model will be implemented eventually. This will allow nodes within a zone to reference elements on other nodes in the zone without specifying the node number. To ease upgrade of parallel models to the shared-namespace paradigm, it is recommended that element names within a zone be unique. If this recommendation is not adhered to, there will be naming conflicts if a model wishes to take advantage of the shared-namespace capability when it becomes available.

21.4.3 Execution (Threads and Synchronization)

The main thread (i.e., flow of control) on each node is that which reads commands from the script file (or keyboard, if the session is interactive). PGENESIS provides limited capabilities for this thread to create new threads on any node. On each node, the threads are pushed onto a stack with the main thread at the bottom of the stack. Only the topmost thread may execute, and when it completes it is popped off the stack so that the next thread down can continue. Threads ready to execute are *not* guaranteed to execute: if the topmost thread is blocked or looping, no ready thread lower on the stack can continue.

An executing thread is guaranteed to run to completion (assuming it does not contain an infinite loop or block on I/O) as long as it executes only local operations, i.e., no operations that explicitly or implicitly involve communication with other nodes. The command descriptions below include specification of local or non-local status. In addition, simulation steps and reset are by definition non-local operations if there is more than one node in the zone. Users are strongly encouraged to use only local operations in child threads whenever possible. Users need to be very careful about thread creation to ensure that *deadlock* (when no thread can continue) does not occur.

PGENESIS provides facilities for blocking and non-blocking thread creation, usually used to execute commands on nodes different from the one on which the script is being executed. ("remote" nodes). When a thread (including the main script) initiates a blocking remote thread (also known as a *remote function call*), it waits until the thread completes before continuing. When a thread initiates a non-blocking remote thread (an asynchronous thread), it continues immediately without waiting for termination of the thread. While a thread is waiting, the node can accept a request for thread creation arriving from any node (including itself). This new thread is pushed on the thread stack and executed, so that the original waiting thread does not continue until the new thread has completed.

Scripts running on different nodes can synchronize via several different synchronization primitives. The simplest, a *barrier*, causes every node to wait at the *barrier* statement until all other nodes have reached that point in the script. There are two types of barriers, one that involves all nodes in a zone, the other involving all nodes in the parallel platform. By default there is an implicit zone-wide barrier before a simulation step is executed, although this can be disabled.

When a script requests that a command be run asynchronously on another node, it initiates a child thread of control on the other node. The child thread runs asynchronously with its parent. The parent can request notification or the child's result when the child completes, and can wait on that notification or result (a "future"), and this is the only way to ensure asynchronous child threads have completed. Threads do not block for child completion before each simulation step, nor at a barrier. It is easy to reach deadlock if the creation and execution of threads are handled carelessly.

If a node initiates several child threads on a particular remote node, these are guaranteed to commence (but not necessarily complete) execution in the order in which they were initiated. A thread is guaranteed to execute eventually as long as no preceding thread (1) enters a loop that only executes local operations, or (2) blocks indefinitely because of deadlock. Once execution of a thread begins, it runs to completion without interruption as long as it only executes local operations.

21.4.4 Simulation and Scheduling

PGENESIS provides the ability to set up a GENESIS message between two elements on different nodes (a remote message), provided the nodes are in the same zone. Data are physically transferred from one node to the next at the beginning of a simulation step. This means that there is no transfer of data between elements on different nodes within a single time step, which has ramifications for the schedule. (The GENESIS Reference Manual contains a description of simulation schedules.) PGENESIS guarantees that execution on a parallel platform will be identical to that on a single processor if and only if there are no remote messages for which the source object precedes the destination object in the schedule. (We assume that every node in a zone has the same schedule.)

21.4.5 Node-Specific Script Processing

Each node executes a single main script common to all nodes. However, it is common to need node-specific script processing. A node is assigned a unique identification which is available to the script it is processing via the *mynode* and *myzone* commands. If execution of script statements is conditional on the node ID, then different nodes can execute different scripts. For example, if the main script (henceforth, *main.g*) executed by all nodes contains this script fragment.

```
early-statements...
if (mynode == 0)
        include node0.g
else
        include nodex.g
end
later-statements...
```

then each node 0 will execute *early-statements*, followed by statements in script *node0.g*, followed by *later-statements*. All other nodes will execute *early-statements*, followed by statements in script *nodex.g*, followed by *later-statements*.

21.4.6 Asynchronous Simulation

The PGENESIS nodes usually operate asynchronously, so instantaneous position in a script varies between nodes. In the preceding script fragment, one node still could be executing *early-statements* while another is already executing *later-statements*. This means, for example, that one node may have completed element creation and be adding messages while another node is still creating elements. If the first node tries to send a message to an element on the second node, it may find that the target element has not been created. Modelers can control this behavior through the judicious use of barriers. A *barrier* is a script statement that must be executed by all nodes before any node can continue past the *barrier* statement. For example, the following main script fragment will guard against the problem of attempting to send a message to an element not yet created:

```
create_elements(arg1, arg2, ...)
barrier
create_messages(arg3, arg4, ...)
```

In this main script fragment, each node creates the required elements, then waits at the *barrier* statement. Only when all nodes have reached the barrier, and therefore created their elements, can any node continue on to create messages.

Some script commands result in implicit synchronization events. For example, by default, the nodes synchronize before executing a simulation step. The *reset* command also causes the nodes to synchronize.

21.4.7 Zones and Node Identifiers

Nodes can be grouped in zones when the simulation is started. Each node is in exactly one zone (by default, every node is in its own zone). The zones form a fixed partition of the parallel platform. The motivation for using zones is to allow different parts of the simulation to run asynchronously (uncoordinated) even during simulation steps. For example, in a parameter search application, one might wish to run many instances of a four-node model in parallel. Each instance uses four nodes that must run synchronously, but the instances need not be coordinated (except at start and finish). Thus, we can run each instance in a separate zone, each zone containing four nodes. Zones are uniquely identified by consecutive integers starting at zero. The nodes within a zone are uniquely identified with a node number (consecutive integers starting at zero).

Node identifiers are of the form "`p.q`" where `p` is the number of the node within zone `q`. Node identifiers are used in commands that expect an element path which may be on a remote node (e.g., *raddmsg* in Sec. 21.4.10) and in remote function calls (see Sec. 21.4.8). The zone specification "`.q`" can be omitted, in which case the zone of the node executing the script is assumed. In network models there is often only a single zone, so that the zone specification can be omitted everywhere. In optimization tasks there is typically only one node in each zone, so that nodes are referred to with "`0.n`".

A script accesses the node number and zone number of the node on which it is running with the *mynode* and *myzone* commands. For example:

```
echo I am node {mynode} in zone {myzone}
```

will print the node and zone numbers on which the script is running.

21.4.8 Remote Function Call

A script running on a node can execute a function or command on another node simply by appending "`@ID`" to the command, where ID is the identification string for the node. We refer to the node on which the script is running as the *issuing* node, and the remote node on which the function or command is executed as the *executing* node for the remote function call. For example, if node 3 in zone 1 executes the remote function call:

```
echo@0 hello from node {mynode} in zone {myzone}
```

then the issuing node is node 3 in zone 1, and the executing node is node 0 in the same zone as the issuer (i.e., zone 1). This command will cause "`echo hello from node 3 in zone 1`" to be executed on node 0 of zone 1. Notice that the evaluation of the commands *mynode* and *myzone* is performed on the issuing node, not the executing node. Argument evaluations are always performed on the issuing node.

Two special keywords can be used with the `@` operator: *all* means all nodes or all zones, depending on context; *others* means all other nodes or zones, depending on context. Here are some examples of how they can be used:

```
step@all                    // step every node in the zone
func@all.all                // call func on every node in every zone
func@3.others               // call func on node 3 in other zones
func@all.3                  // call func on every node in zone 3
```

In addition, node identifiers can be composed into a list separated by commas. For example

```
func@1,3,5                  // call func on nodes 1, 3 and 5
```

This is used in the network model in Sec. 21.6.

The remote function call examples shown in this subsection are *synchronous*. The issuing node suspends execution of the script containing the remote function call statement until the executing node has completed the function call and returned the result. This is shown schematically in Fig. 21.1. In this figure we also show the execution of the script on node 1 as a solid line. Conceptually, there are just two threads of control active at any time. During the execution of the remote function call on node 1, both threads reside on node 1. In PGENESIS there is exactly one thread of control per node, as long as the *async* command is not executed, but by using remote function calls, some nodes may have no threads resident, and other nodes may have more than one resident thread.

PGENESIS users should be aware that the threads of control are not full, independent threads, able to suspend and resume arbitrarily. They are implemented in a stack-based system so that only the thread at the top of the stack can execute. When it completes it is popped off the stack and the preceding thread resumes. (It may immediately suspend again, but it is given the chance to resume.)

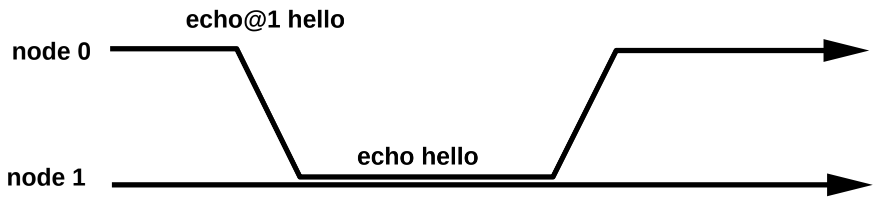

Figure 21.1 Synchronous Remote Function Call. Node 0 issues the command "`echo hello`" to node 1, and suspends. When node 1 has executed the command and returned the result to node 0, node 0 continues.

21.4.9 Asynchronous Remote Function Call

It is also possible for a node to issue a remote function call asynchronously using the *async* command. We recommend that only experienced users of PGENESIS use this command. A simple example is:

```
async echo@1 hello
```

In this case, the issuing node sends off the request for the command to be done to the executing node (1 in this example) and continues processing its script without waiting for the executing node to complete (or even start) the command. This is shown schematically in Fig. 21.2. Notice that after the remote function call has been issued by node 0, and until the result is received, there are three separate threads of execution: the parent on node 0 which continues; the child on node 1 which executes concurrently with the parent; and the original thread on node 1. Every use of the *async* command introduces one or more additional threads of control beyond the initial single thread per node with which PGENESIS starts.

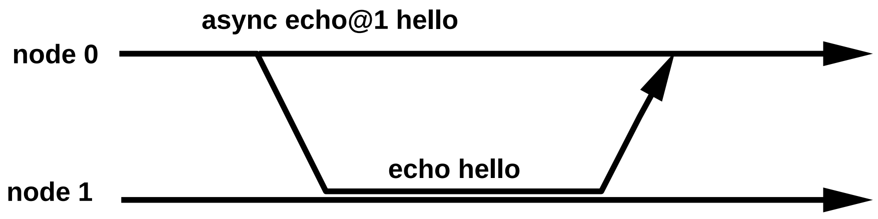

Figure 21.2 Asynchronous Remote Function Call. Node 0 issues the command "`echo hello`" to node 1, and continues executing. When node 1 has executed the command it returns the result to node 0, which does not use it.

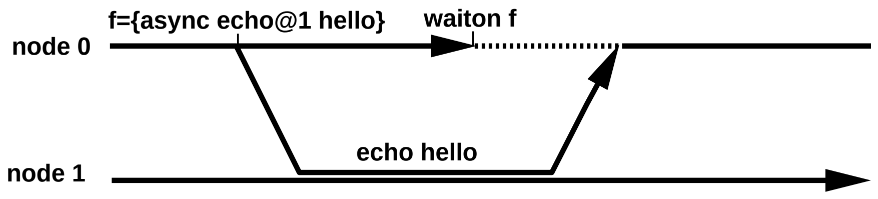

Figure 21.3 Completing an Asynchronous Operation. "`echo hello`" to node 1, and continues executing. When node 1 has executed the command it returns the result to node 0, which does not use it.

Every asynchronous operation issued from a node returns a result to the issuing node upon completion. A script can issue an asynchronous command, do some further processing, and then wait for the command to complete:

```
int future
future = {async some-function@1 args... }
some-useful-function args...
waiton future
```

The execution diagram for this script fragment is shown in Fig. 21.3.

The *future* variable is a handle returned by the *async* command that is passed to the *waiton* command. The script then suspends until the asynchronous operation completes (the future is satisfied). This "join" operation results in a reduction in the number of threads of control. The number of threads that terminate is exactly the same as the number of threads that were created by issuing the asynchronous operation.

A special form of the *waiton* command allows a node to wait for all its outstanding asynchronous operations to complete:

```
waiton all
```

This form of the *waiton* command reduces the number of threads of control that originated on the current node to exactly one. If every node executes this command followed by a barrier, then there will be exactly one thread of control per node when the barrier is satisfied.

21.4.10 Message Creation

For connecting elements that reside on the same node, the usual GENESIS commands (*addmsg* and *volumeconnect*) are available. However, PGENESIS also supports the creation of messages between elements that reside on different nodes. *raddmsg* allows one to create a message between an element on the node where the *raddmsg* is executed (the “local” node) and an element on another node (the “remote” node). For example, to create a PLOT message from */cell/soma* on node 1 to a graph on node 0, the following should be executed on node 1.

```
raddmsg /cell/soma /data/voltage@0 PLOT Vm *volts *red
```

One can also create this message from node 0 by using the remote procedure call mechanism:

```
raddmsg@1 /cell/soma /data/voltage@0 PLOT Vm *volts *red
```

There is also an *rvolumeconnect* command (analogous to *volumeconnect*) that is illustrated in the network example in this chapter (Sec. 21.6). It is currently not possible to delete remote messages.

21.5 Running PGENESIS

To run PGENESIS, you should first make sure that it has been installed, following the directions in the distribution. If PGENESIS was not included with your GENESIS distribution, you may obtain it from the PGENESIS web site (Goddard and Hood 1996), along with the latest version of the documentation.

We start with one of the simplest PGENESIS scripts — a parallel version of the classic “hello, world” program. We’ll examine this script line by line, adding some line numbers that aren’t present in the actual script:

```
1 paron -nodes 3 -parallel
2 echo@0 hello from node {mynode}
3 barrier
4 paroff
5 quit
```

1. `paron -nodes 3 -parallel` tells PGENESIS to initialize the parallel capability running 3 nodes in total, in the synchronized form (i.e., all in a single zone). PGENESIS will start up two more nodes executing the same script.

2. `echo@0 hello from node {mynode}` tells the node to issue an echo command to node 0 which prints the *issuing* node number.

3. `barrier` causes all nodes to wait until the barrier is reached. The purpose of this barrier is to make node 0, which in this script is reporting which nodes are running, wait until all the nodes have had their echo command printed.

4. `paroff` causes each node to flush its standard output and standard error buffers (so that output is written out) and then enter a barrier. Thus, by the time this command completes, all nodes will have flushed their buffers.

5. `quit` exits to the UNIX prompt.

In summary, the effect of this script is that output similar to the following will appear on node 0.

```
hello from node 0
hello from node 2
hello from node 1
```

Because of the parallel nature of the code, the particular order of these messages is not deterministic, and so you may sometimes find, for example, that node 1's message appears first.

21.5.1 The *pgenesis* Startup Script

PGENESIS is usually run by executing the *pgenesis* script. This script performs some checks on the execution environment, starts the PVM daemon on the appropriate machines, and runs the appropriate initial executable for PGENESIS. To try this out, locate the *pgenesis* script and put it on your PATH. Henceforth we assume that typing "`pgenesis`" results in this script being run. Put the "hello, world" script listed above into a file, say, "`hello.g`". Now you should be able to execute the "hello, world" script with:

```
pgenesis hello.g
```

The full set of flags for this script is described with the PGENESIS hypertext documentation (Goddard and Hood 1996), which is also included in the PGENESIS distribution. In addition to the usual flags available for GENESIS, some of those interpreted by PGENESIS include:

-config *filename* The specified file should contain a list of hosts to use to run the scripts (e.g., "`axp0 axp1 axp2`"). Names should be separated by blanks or newlines.

-debug *mode* Run the workers in their own separate windows to allow debugging at either the GENESIS script level or at the C code source level. Not all modes are supported on all platforms. If you specify an unsupported mode the *pgenesis* shell script will select an alternative. Valid modes are:

1. **tty** — run the workers in individual windows but not under any C debugger.
2. **dbx** — run the workers in individual windows under the control of dbx.
3. **gdb** — run the workers in individual windows under the control of gdb running inside emacs.

-nox Run a version of the PGENESIS executable that does not have the XODUS libraries loaded — this is smaller, starts up faster, and does not require you to be running X windows. Note this only applies to the first node. Subsequent nodes are started with the *paron* command in the script, which can specify an executable.

-v Run in verbose mode.

-help Print text describing all the flags.

21.5.2 Debug Modes

The tty debug mode supported by the *pgenesis* script can be useful for debugging parallel scripts. The other two debug modes, which run PGENESIS nodes under the dbx or gdb debuggers, are most useful for advanced users who have interfaced their own C code with GENESIS.

To illustrate the use of the tty mode, and some of the functionality provided by the PGENESIS extensions to the script language, we will show you how to do some interactive programming of PGENESIS. This is not the way to develop parallel scripts, but it will give some insight into PGENESIS, and it can be useful in debugging parallel scripts.

First create a script *debug.g* containing this single command, which tells PGENESIS to run with 3 nodes and with the `silent` level at 0, i.e., so that all the usual banner and error messages are printed:

```
paron -nodes 3 -silent 0 -parallel
```

Now run PGENESIS with:

```
pgenesis -debug tty debug.g
```

This should start up a node that spawns two new nodes, so that there are three PGENESIS nodes running. Each of the spawned nodes appears in its own window, and after startup all nodes show a prompt.

Execute a remote function call from any of the nodes:

```
echo@all hello from {mynode}
```

This causes a “hello” message to appear in each node window.

Now try a barrier. Type “`barrier`” to the prompt in each of the node windows. Notice that the prompt does not reappear in any window until the *barrier* command has been issued on each node.

Now exit. Type “`quit@all`” to any of the prompts. The spawned node windows will disappear and, after cleaning up, the original node will exit to the UNIX prompt.

21.6 Network Model Example

[If you are not interested in distributing a network model over multiple nodes, this section may be skipped.]

This section illustrates how the *Orient_tut* example in Chapter 18 can be parallelized. Recall that that network has an array of retinal cells whose axons make contact with two populations of V1 cells. In parallelizing a network model, the most critical decision is how to decompose the network, i.e., how to distribute the cells amongst PGENESIS nodes. The goal is to minimize the number of synapses crossing node boundaries while maximizing the axonal delay of those synapses that do cross node boundaries. In *Orient_tut*, the connectivity is feedforward from the retina to V1 with some spatial divergence. A simple but effective decomposition is to divide the *x*-dimension of the retina and V1 populations amongst processors, as shown in Fig. 21.4.

In explaining this example, we do not provide the full scripts. These reside in the *Scripts* directory of the PGENESIS distribution. Instead, we show the important features of the parallelization, particularly how the simulation is set up and controlled and what is modified from the serial version described in Chapter 18. Nor do we discuss the issues involved in generating an XODUS display, which are covered later in Sec. 21.8.

21.6.1 Setup

In setting up the simulation, we will use one node to control the simulation, and the remaining nodes to run the slices of the model. Thus, n slices will require $n+1$ nodes. These will run in a single zone (i.e., stepping is synchronized), with node 0 controlling, and nodes $1,\ldots,n$ (the workers) running the slices. The main global variables used are:

```
int n_slices = 5        // number of slices
int retina_nx = 10      // retina has 10 x 10 cells
int retina_ny = 10
int v1_nx = 5           // each V1 population has 5 x 5 cells
int v1_ny = 5
int sim_steps = 1000    // when sweep is requested, we will do 100 steps
int i_am_control_node, i_am_worker_node // Booleans indicating the function
int slice                       // what slice this node holds
```

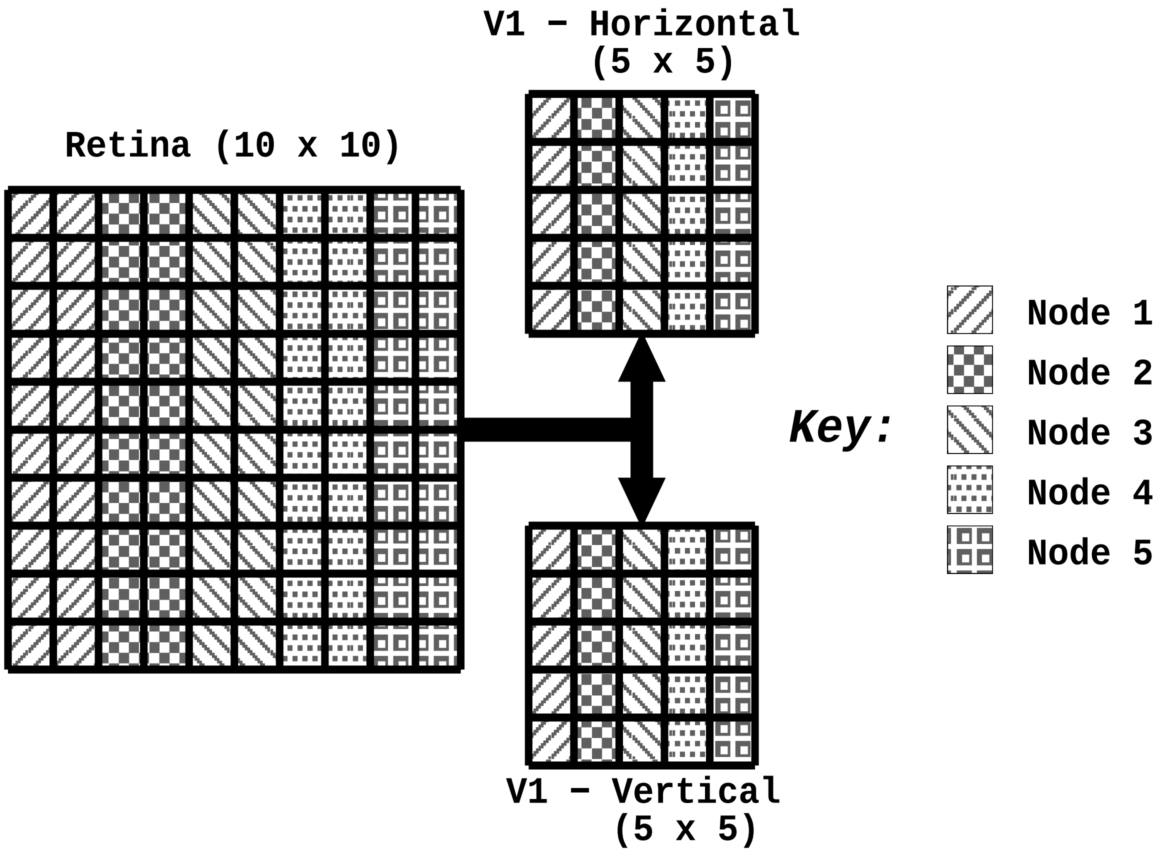

Figure 21.4 Slice decomposition of the retinal and V1 cells onto a set of 5 nodes.

```
str workers                     // a list of nodes holding slices
int i                   // iterator
int n_nodes = n_slices + 1
```

The setup part of the controlling script, executed by all nodes, is:

```
workers = "1"
for (i=1; i<=n_slices; i=i+1); workers=workers @ "," @ {i}; end
paron -parallel -silent 0 -nodes {n_nodes}
setfield /post msg_hang_time 100000
i_am_control_node = {mynode} == 0
i_am_worker_node =  {mynode} > 0
randseed {mynode * 347}
if (i_am_control_node); setup_control
else; create_slice_cells; end
```

```
10 barrier
11 if (i_am_worker_node); connect_retinal_slice; end
12 reset
```

Again, this listing has added line numbers that are not present in the actual script. This script creates a string containing the list of worker nodes (lines 1–2), then initializes PGENESIS (line 3) with *n_nodes* nodes. Line 4 sets the timeout to a very large value so that worker nodes, which wait at a barrier (line 23) will not timeout. Lines 5–6 set Booleans indicating the function of the node. Recall that every node executes this script and *mynode* returns the number of the node. Line 7 initializes the random number generator with a different value for each node, to avoid identical random number sequences on the different nodes. If the node is the control node, *setup_control* is called to initialize the control process (line 8). If it is a worker node, the slice cells are created (line 9). The purpose of the barrier in line 10 is to ensure that all the cells have been created before any node attempts to make connections (line 11). Although the control node makes no cells, it must participate in the barrier because a barrier always includes all nodes in a zone. After connections have been made, all nodes reset (line 12). The remainder of this script fragment (lines 13–24) appears in the following section on simulation control.

The code to create the cells in slices is only marginally different from that for the serial version (in *Orient_tut/retina.g*). For example, the retinal slice is created with:

```
createmap /library/rec /retina/recplane {REC_NX / n_slices} {REC_NY} \
    -delta {REC_SEPX} {REC_SEPY}   \
    -origin {-REC_NX * REC_SEPX / 2 + \
         slice * REC_SEPX * REC_NX / n_slices} {-REC_NY * REC_SEPY / 2}
```

Here, the globals *slice* and *n_slices* are used to determine how many cells to create and what their spatial locations are. As described in Sec. 18.6, the *createmap* commands perform calculations that create the V1 populations on two-dimensional grids.

The connections from retinal to V1 cells are made with calls to *rvolumeconnect*. This command is just like *volumeconnect* except the destination path indicates on which nodes to look for the destination elements. Here we specify all the slice nodes. Although we could compute exactly which nodes should have the appropriate destination cells, it is easier to just ask PGENESIS to check everywhere. However, this does result in unnecessary communication between nodes during setup. In this example it is not a significant factor, but in more complex examples, especially with many nodes, it could be much more efficient to do this computation in the script. The connections to the V1 horizontal cells, for example, are made thusly:

```
rvolumeconnect /retina/recplane/rec[]/input \
  /V1/horiz/soma[]/exc_syn@{workers}   \
  -relative                         \
  -sourcemask box -1 -1 0  1 1 0    \
  -destmask box {-2.4 * V1_SEPX} {-0.6 * V1_SEPY} {-5.0 * V1_SEPZ} \
                { 2.4 * V1_SEPX} { 0.6 * V1_SEPY} { 5.0 * V1_SEPZ}
```

After the connections have been established, we can modify the axonal delays and synaptic weights with *rvolumedelay* and *rvolumeweight* which are analogs of the *volumedelay* and *volumeweight* commands in GENESIS. For example:

```
rvolumedelay /retina/recplane/rec[]/input -radial {CABLE_VEL}
rvolumeweight /retina/recplane/rec[]/input -fixed 0.22
```

21.6.2 Simulation Control

Lines 13-24 of the main script fragment, continued from above contain the crucial code for controlling the simulation:

```
13 if (i_am_control_node)
14   if (batch)
15     autosweep horizontal
16     barrier 7
17     paroff; quit
18   else
19     echo issue commands, then quit@all to terminate
20   end
21 else
22   barrier 7 100000
23   paroff; quit
24 end
```

Worker nodes simply sit at a barrier (line 22), ID number 7 (chosen simply for uniqueness), waiting for commands from the control node for a maximum of 100,000 seconds. If the simulation is running interactively, the control node prints a message (line 19) and then the script terminates, returning to the PGENESIS prompt. If the script is running in batch mode, the control node executes a simulation (line 15), then satisfies the barrier at whcih the workers are waiting. This allows the workers to continue on and quit (line 23). The control node similarly quits (line 17).

The function *autosweep* calculates the parameters for sweeping a bar across the retina and then steps the simulation on all nodes for the appropriate number of steps, setting the appropriate input in the retina before each step.

```
1 function autosweep
```

```
2   init_bar_params
3   for (i = 0; i < sim_steps; i = i + 1)
4     compute_bar_corners
5     setfield@{workers} /retina/recplane/rec[x>{x1}][y>{y1}] \
6       [x<{x2}][y<{y2}]/input rate {rate} -empty_ok
7     step@all
8   end
9 end
```

init_bar_params (line 2) is a script function not shown here that initializes the computation of the sweeping bar. Prior to each simulation step (line 7), the corners of the bar are computed with *compute_bar_corners* (line 4), not described further. This sets the global variables *x1*, *y1*, *x2*, *y2*, which are used in wildcard tests in the *setfield* command (lines 5–6), which actually sets the input in the retinal cells. Recall that *autosweep* is only called on the control node (see line 15 at the beginning of this subsection). It issues the *setfield* command for each node, so that all slices of the retina are properly initialized for the step. Notice the added flag "`-empty_ok`". This flag is an addition to the *setfield* command which tells it that it is *not* an error if no elements match the wildcard specification. In the sliced up simulation, some nodes may not contain any retinal cells that are inside the spatial extent of the bar. It is simpler to allow *setfield* to accept an empty wildcard list of elements than to compute, for each step, exactly which nodes have relevant retinal cells and which don't.

21.6.3 Lookahead

In network models, axonal delays are typically one or two orders of magnitude greater than the simulation time step for processes within a single cell. A spike generated at simulation time T need not be delivered to a destination cell until time $T + L$, where L is the axonal delay. This allows simulation nodes to operate in a loosely synchronized fashion, some being ahead of others, and it allows nodes to continue updating their cells while incoming spikes are in transit over the physical medium (e.g., Ethernet) that connects the CPUs. The amount of simulation time by which node A can get ahead of node B and still be sure it has not missed any spikes is known as the *lookahead* of A with respect to B.

In PGENESIS lookahead is controlled with three commands: *setlookahead*, *getlookahead* and *showlookahead*. The lookahead of node A with respect to node B is the minimum delay on all data paths from B to A, i.e., the minimum axonal delay over all the connections from B to A. If there are no axonal paths from B to A, the lookahead is infinite because A's activity does not depend on B's. If there are non-spike messages, lookahead is dt because those messages deliver data on the next time step. In addition, PGENESIS must be instructed to execute steps asynchronously with the command:

```
setfield /post sync_before_step 0
```

/post is the element that provides access to PGENESIS configuration fields. The variable *sync_before_step* is a Boolean indicating whether PGENESIS should synchronize nodes in a zone before a simulation step. By default it has value 1, so that nodes synchronize. To use lookahead, synchronization before stepping must be turned off.

By default, the lookahead is set to the time step, since every PGENESIS node always delivers spikes on the next time step. To set the minimum lookahead of a node with respect to all other nodes to 10 *msec*:

```
setlookahead 0.01
```

To set the minimum lookahead of a node with respect to, e.g., node 3 to 10 *msec*:

```
setlookahead 3 0.01
```

To find the lookahead of this node with respect to, e.g., node 4:

```
getlookahead 4
```

To view the lookahead of this node with respect to all other nodes:

```
showlookahead
```

It is wise to partition a model in such a way as to maximize lookahead between every pair of nodes. Techniques for automated partitioning are under investigation, but intelligent choice of partition by the modeler will remain a critical aspect of efficient simulation for some time to come.

21.7 Parameter Search Examples

[If you are not interested in parameter search, this section may be skipped.]

As mentioned in Sec. 7.4.1, GENESIS contains objects and commands for performing automated parameter searches, in order to estimate a set of model parameters that gives the best fit of the model behavior to the results of experiments. To demonstrate the use of PGENESIS for parameter estimation, we have constructed an example that performs a simple genetic search. The search problem in this example is quite trivial — we are trying to find values of parameters a and b that minimize $min(|a-1|+|b-1|,|a+1|+|b+1|)$. This function has two minima, one at $(-1,-1)$ and one at $(1,1)$, so our search procedure should find one of these. A more realistic parameter search in the neuroscience domain would have parameters that represent, for example, channel conductance densities in a cell model. The evaluation of the parameter set would be obtained by first running a neural simulation for, e.g., 100 milliseconds, and then comparing the simulation results (e.g., voltages or spike times) with experimental data, and generating a numerical value to represent the goodness of the match. However, we have substituted here a simple function evaluation so that we can more cleanly illustrate a method for doing this type of search using PGENESIS. We go through the entire script in this section, starting with the high-level design and gradually refining the implementation in script commands.

In performing the genetic search, we keep a fixed-sized population of individuals "alive." We randomly pick an individual from this population and mutate its representation with a certain probability. We then evaluate the new individual and, if it is better than some other, we replace the worst-evaluated individual in the population with this new individual. To allow for parallelism, we evaluate multiple new individuals simultaneously, and only remove existing individuals in the population as new individuals with superior evaluations are found.

In this example, we represent each parameter (out of a total of 2 parameters per individual) as a 16-bit string. The floating point value that this bit-string represents is determined by linearly mapping the 16-bit integer with range [0,65535] into the range $[-32.768, 32.767]$. When constructing a new individual from an old one, we mutate each of the bits with probability 0.02. We evaluate each individual by computing a trivial function over the parameters. In a real-world parameter search, this step would be the most time-consuming, since it would involve running a neural simulation. The main PGENESIS script follows, with each statement numbered.

```
1 paron -farm -silent 0 -nodes {n_nodes} -output o.out -executable nxpgenesis
2 echo@0 node {mytotalnode} started
3 barrierall
4 if ({mytotalnode} == 0)
5       search
6 end
7 barrierall 7 1000000
8 paroff
9 quit
```

This is the top-level execution path for the nodes. The *paron* command (line 1) starts up the nodes in farm mode; i.e., each node is in its own zone, with output going to file *o.out*, and using the non-XODUS executable for spawned nodes since they do no graphical display. Line 2 uses a remote function call to print a startup message on node 0. Line 3 causes all nodes to synchronize here. *barrierall* is used because the nodes are in separate zones — recall that the *barrier* command synchronizes the nodes in a zone, whereas *barrierall* synchronizes all the nodes in every zone. Line 5 is only executed by the global zero node (*mytotalnode* gives the global node number, and *mynode* gives the node number within the zone). Line 5 is a call to the function *search*, described below, which conducts the parameter search. The non-zero nodes continue to line 7, which causes them all to wait at the barrier (with ID 7 and timeout 1 million seconds). The zero node does not reach this barrier until the call to *search* in line 5 returns, i.e., until the search is complete. At that point, the zero node satisfies the barrier and all nodes flush their buffers and synchronize (line 8) then exit (line 9). In summary, the non-zero nodes sit at a barrier while the zero node conducts the search; then all nodes exit.

We use a number of global variables in the script:

```
int n_nodes = 4               // number of PGENESIS nodes to use
int individuals = 1000        // number of individuals to evaluate
int population = 100          // size of population to maintain
float bit_mutation_prob = 0.02 // mutation probability
float min_fitness, max_fitness // current worst/best fitness values
int actual_population = 0     // size of current population
int least_fit, most_fit       // indices into fitness data structure
int free_index = 0            // index in farm of free worker
int bs_a, bs_b                // bit string representation of parameters
```

While the non-zero nodes are sitting at a barrier, they can process incoming remote function calls and other requests from other nodes. The zero node conducts the search by issuing remote function calls to the non-zero nodes to evaluate individuals and return fitness values. The function that executes this search is:

```
1  function search
2    int   i
3    init_search; init_farm
4    for (i = 0; i < individuals; i = i + 1)
5      if (i < population); init_individual
6      else; mutate_individual {rand 0 actual_population}; end
7      delegate_task {i} {bs_a} {bs_b}
8    end
9    finish; echo; echo "Finished search at " {getdate}; print_best
10 end
```

In this function, line 3 initializes data structures used in conducting the search (*init_search*) and managing the farming out of tasks to nodes (*init_farm*). The loop in lines 4–8 farms out the evaluations to the nodes. Lines 5 and 6 select parameter values for the evaluation. An initial population is selected randomly (*init_individual*), after which new individuals are derived from the existing population by mutation (*mutate_individual*). Line 7 sends the task to the worker (*delegate_task*) using parameter values *bs_a* and *bs_b*. Line 9 ensures the search has completed (*finish*) and then prints the best match (*print_best*). These functions are described below.

```
function init_search
  create neutral /gs; disable /gs
  create neutral /gs/population
  addfield /gs/population a_value; addfield /gs/population b_value
  addfield /gs/population fitness
  createmap /gs/population / {population} 1
end

function init_farm
  create neutral /farm; disable /farm
  create neutral /farm/free; addfield /farm/free value
  createmap /farm/free / {n_nodes-1} 1
  for (i=0; i<{n_nodes-1}; i=i+1); setfield /free[{i}] value {i+1}; end
  free_index = n_nodes - 2;
end
```

These two functions create (with *createmap*) and initialize the data structures used to control the search (*/population*) and the farming out of tasks to nodes (*/free*). Each individual in the */population* vector has fields for the values of the two parameters and the fitness those parameters provide. The entries in the *free* vector have one field, the zone number of a node that does not currently have a task assigned. The *free_index* variable is an index into this vector of the last entry that contains a free node.

```
function init_individual
    bs_a = {rand 0 65536}; bs_b = {rand 0 65536}
end

function mutate (v)
    int v, i, b = 1
    for (i = 0; i < 16; i = i + 1)
        if ({rand 0 1} < bit_mutation_prob); v = v ^ b; end
        b = b + b
    end
    return {v}
end

function mutate_individual (chosen)
    int chosen
    bs_a = {mutate {getfield /population[{chosen}] a_value}}
    bs_b = {mutate {getfield /population[{chosen}] b_value}}
end
```

The three functions above generate a new individual. *init_individual* generates random parameter values. *mutate_individual* generates a parameter values by mutating the *chosen* individual with *mutate*.

```
function delegate_task
```

```
2   while (1)
3     if (free_index >= 0)
4       async worker@0.{getfield /free[{free_index}] value} {bs_a} {bs_b}
5       free_index = free_index - 1;
6       return
7     else; clearthreads; end
8   end
9 end

function finish
  while (free_index < n_nodes - 2); clearthreads; end
end
```

These two functions control the assignment of tasks to nodes and the gathering of results. *delegate_task* waits until there is a free node, then issues an asynchronous remote function call to the worker to evaluate the current parameterization (*bs_a*, *bs_b*). Line 3 tests for a free node. If there is one, the task is assigned (line 4), the node is removed from the free set (line 5), and the function returns (line 6). If there is no free worker, worker responses are checked for (line 7) and the loop iterates (line 2) until one is found.

This function prints out the best match found during the search:

```
function print_best
    float a, b
    a = ({getfield /population[{most_fit}] a_value} - 32768.0) / 1000.0;
    b = ({getfield /population[{most_fit}] b_value} - 32768.0) / 1000.0;
    echo "Best match with a = " {a} ", b = " {b} ", fitness = " {max_fitness}
end
```

There is one other function which executes on the node that controls the search (zone 0). When a node has completed an evaluation, it issues a remote function call for the controlling node to report the result. It calls:

```
1  function return_result (node, bs_a, bs_b, fit)
2    int node, bs_a, bs_b
3    float fit
4    if (actual_population < population)
5      least_fit = actual_population; min_fitness = -1e+10;
6      actual_population = actual_population + 1
7    end
8    if (fit > min_fitness)
9      setfield /population[{least_fit}] fitness {fit}
10     setfield /population[{least_fit}] a_value {bs_a}
11     setfield /population[{least_fit}] b_value {bs_b}
12     if (actual_population == population); recompute_fitness_extremes; end
13   end
```

```
14   echo " " {fit} -n
15   free_index = free_index + 1
16   setfield /free[{free_index}] value {node}
17 end
```

return_result adds the individual (lines 9–12) to the population if either there is room (line 4), or if the individual has fitness greater than some individual currently in the population (line 8). It prints the fitness (line 14) and puts the node in the */free* vector (lines 15–16). Notice in line 12 that if the population is full, then the individuals with minimum and maximum fitness are computed with *recompute_fitness_extremes*, shown below.

```
function recompute_fitness_extremes
  min_fitness = 1e+10; max_fitness = -1e+10
  for (i = 0; i < actual_population; i = i + 1)
    if ({getfield /population[{i}] fitness} < min_fitness)
      least_fit = i; min_fitness = {getfield /population[{i}] fitness}
    end
    if ({getfield /population[{i}] fitness} > max_fitness)
      most_fit = i; max_fitness = {getfield /population[{i}] fitness}
    end
  end
end
```

Since this function is called by *return_result*, it also executes on node 0, which controls the search. The evaluating node code is much simpler in this example because our evaluation function *evaluate* is trivial. In a real application, the evaluation would be computed by running a GENESIS model and comparing its output with experimental data. The two functions that execute on the evaluating node in our example are:

```
function worker (bs_a, bs_b)
  int bs_a, bs_b
  float a, b
  float fit
  a = (bs_a - 32768.0) / 1000.0; b = (bs_b - 32768.0) / 1000.0;
  fit = {evaluate {a} {b}}
  return_result@0.0 {mytotalnode} {bs_a} {bs_b} {fit}
end

function evaluate (a, b)
    float a, b, match, fit
    match = {min {{abs {a-1}} + {abs {b-1}}} {{abs {a+1}} + {abs {b+1}}}}
    if (match != 0.0); fit = 1.0/{sqrt {match}}
    else; fit = 1e+9; end
    return {fit}
end
```

worker is the function called by the controlling node to execute an evaluation. It converts the bit string representation of the parameters to floating point values, computes the fitness with a call to *evaluate*, and returns the result to the controlling node (0.0) using a remote function call. In our example, *evaluate* computes a match value that has two minima at $(1,1)$ and $(-1,-1)$, and returns a fitness value that is the inverse of the square root of the match.

21.8 I/O Issues

Parallel simulations can often benefit from visualizations with XODUS during development, and often will require large input and/or output files in full scale production runs. Modelers should be aware that the way these I/O issues are dealt with can have a considerable impact on performance.

PGENESIS includes a capability to allow multiple nodes to display on the same XODUS widget so that, for example, a single **xview** can be used to show activation of all the cells in a distributed V1 layer. In serial GENESIS there are several ways to set up input to an **xview** element, described in Chapter 18 and in the documentation for **xview** in the GENESIS Reference Manual. However, the one we must use for internode communication in PGENESIS is to set up remote messages from a source element to a destination **xview** element. This is done by using the PGENESIS *raddmsg* command.

If every node were to set up COORDS and VAL*n* messages independently, the VAL messages could easily get associated with the wrong COORDS messages, depending on the order in which the particular add message requests were handled. To deal with this difficulty, the standard GENESIS **xview** object has been extended in PGENESIS to allow IVAL*n* messages to be associated with a particular ICOORDS message. The user does this by choosing an integral index with each message that is set up, and passing it as the first parameter of the ICOORDS and IVAL*n* messages. IVAL1 through IVAL5 messages will be associated with ICOORDS messages having the same index. For an example of this, see the example script *Scripts/par_io/par_view.g* in the PGENESIS distribution.

PGENESIS also includes a capability for writing a single disk file from multiple nodes. For disk output in serial GENESIS, it is typical to create an **asc_file** element and then set up a SAVE message that will cause a value to be written to a file on every time step. In PGENESIS it is possible to add such messages from elements on various nodes. However, there is no guarantee of order for the normal **asc_file** object, so in PGENESIS the **par_asc_file** object is provided. When SAVE messages are set up, the first parameter is an integral index that is used to maintain a fixed ordering among all of the various incoming messages to the **par_asc_file** element. This integral index should be unique and in the range from 0 to the number of incoming messages minus 1, inclusive. Serial GENESIS uses the **disk_out** object for writing array data to a file in an efficient binary format. We have similarly provided a corresponding **par_disk_out** object that takes an added integral index parameter. The *Scripts/par_io/par_out.g* file in the PGENESIS distribution illustrates the use of this object.

Both of these extensions for I/O support require that information flow in the form of PGENESIS messages from the source node to the destination node (which holds the **xview**, **par_asc_file**, or **par_disk_out** element). If you are doing very large amounts of I/O from many nodes, the destination node would likely become a simulation bottleneck. In those situations, it would likely be advantageous to consider a solution where each node was doing its I/O to and from files on the local disk, rather than using the above mechanisms.

21.9 Summary of Script Language Extensions

21.9.1 Startup/Shutdown

To use any of the capabilities of the parallel library, one must first start up the library. This will also spawn the requested number of worker nodes on architectures that support process-spawning.

paron	Starts up the parallel library.
paroff	Shuts down the parallel library.

There are several commands for obtaining configuration information:

mynode	Number of this node in this zone.
nnodes	Number of nodes in this zone.
myzone	Number of this node's zone.
nzones	Number of zones.
ntotalnodes	Number of nodes in all zones.
mytotalnode	Unique number over all zones for this node.
mypvmid	Task identifier used by PVM for this node.
npvmcpu	Number of CPUs used by PVM in the parallel machine.

The ability to run parallel threads can be turned on or off (the default is on) with the related commands:

threadson	Re-enables parallelism.
threadsoff	Disables parallelism.
clearthreads	Process all queued requests from other nodes.
clearthread	Process one queued request from another node.

21.9.2 Adding Messages

It is possible to create arbitrary messages between elements on different nodes using the *raddmsg* command:

raddmsg	Adds message between the listed source elements and the listed destination elements (which may be designated to be on other nodes by means of the "@" notation).

The following routine displays internode messages correctly (and suppresses the display of the postmaster messages used to implement the internode messages).

rshowmsg	Shows the messages (intranode and internode) associated with a given element.

21.9.3 Synaptic Connections

There are several routines that allow one to set up multiple synaptic connections across nodes. They are analogs of the normal GENESIS routines for setting up synapses.

rvolumeconnect	Connects one group of elements in a volume to another, using source and destination element lists and masks.
rvolumedelay	Sets delays of a group of synapses receiving input from a list of presynaptic elements in a volume.
rvolumeweight	Sets weights of a group of synapses receiving input from a list of presynaptic elements in a volume.

21.9.4 Remote Command Execution and Synchronization

command@nodelist	Executes command on specified nodes synchronously (i.e., does not return until remote commands have completed and returned result).
async command@nodelist	Executes command on specified nodes asynchronously (i.e., returns integer "future" without waiting for result).
waiton	Wait for completion of a specified *async* command.
barrier	Wait for all nodes in my zone to reach this point.
barrierall	Wait for all nodes in all zones to reach this point.

21.9.5 PGENESIS Objects

postmaster	One postmaster (*/post*) is created per node by the *paron* command to manage internode synchronization and communication.
par_asc_file	Analogous to **asc_file**, except uses an ordering index.
par_disk_out	Analogous to **disk_out**, except uses an ordering index.

21.9.6 Modifiable PGENESIS Parameters

Several parameters of PGENESIS can be modified by the user by setting field values in the */post* element. These fields, and their meanings, are:

1. *sync_before_step*. A Boolean indicating whether nodes in a zone synchronize before a simulation step. The default value is 1 (true). Asynchronous simulation is required for the lookahead optimization and may be faster even without lookahead.

2. *remote_info*. A Boolean indicating if information about messages between nodes should be kept for display with *rshowmsg*. This is an overhead that could be dispensed with for mature models. The default value is 1.

3. *perfmon*. A Boolean indicating if performance statistics should be gathered. This is a feature under development, explained in the PGENESIS documentation. The default value is 0.

4. *msg_hang_time*. A floating point value indicating how many seconds PGENESIS should wait before timing out on remote operations. The default value is 120.0 seconds. If debugging interactively, it is often useful to set this to a very large value so that one does not have to worry about the timing out of other nodes.

5. *pvm_hang_time*. A floating point value indicating how many seconds before timing out that PVM internal operations should wait. When a PGENESIS node is waiting for some message it expects, it will print a message followed by dots every *pvm_hang_time* seconds. The default value is 3.0 seconds.

6. *xupdate_period*. A floating point value indicating the number of seconds between a PGENESIS node's requests that X events be processed, when it is waiting for some expected message. High values can cause poor response from XODUS widgets. Low values can have an adverse impact on performance — but you shouldn't be using XODUS if you want performance. The default value is 0.01 seconds.

21.9.7 Unsupported and Dangerous Operations

It is extremely easy to reach deadlock in parallel programs; one way to reduce the chances of this is the frequent use of barriers and sparse use of asynchronous commands. However, barriers can be expensive to execute and can reduce parallelism, so they should be placed judiciously in scripts.

The serial GENESIS *stop* command should be used only with extreme care in zones containing more than one node. PGENESIS executes an implicit barrier before performing a simulation step. If any nodes enter the barrier, then all nodes must, otherwise deadlock will result. It is very difficult to satisfy this requirement when the *stop* command is issued.

Issuing *step* commands must be done with care. Since the *step* command executes an implicit barrier, failure to observe the following rule can result in deadlock. The two safe methods to issue step commands are:

1. *step* commands are issued exclusively locally (i.e., no use of the @ operator with *step*).
2. remote simulation *step* commands (e.g., `step@all`) are issued by at most one node in a zone.

21.10 Exercises

1. Modify the "hello, world" program in Sec. 21.5 to have node 0 print "`hello, world`" on both nodes 1 and 2. Use the *pgenesis* script in the "`-debug tty`" mode to observe that the output is correctly produced on the worker nodes. Now, change the *paron* statement to include "`-output workers.out`" so that their output is redirected into a file. When doing this, be sure to invoke *pgenesis* without the "`-debug tty`" flag or that will override the file redirection.

2. Create a very simple network model on two nodes using the neural model provided in *Scripts/simple*. Incorporate this by *include*'ing the neuron.g file in that directory. Write a script to create neuron A on node 0 and neuron B on node 1. Create a synapse from A to B using the *raddmsg* command. Verify that the model works by creating a display element on both node 0 and node 1 using the *create_display* function. Manually fire neuron A on node 0 and watch that the neuron B on node 1 fires in succession.

3. Partition the network model example of Sec. 21.6 so that all retinal nodes reside on node 1, and all V1 cells on node 2. Display the V1 horizontal and V1 vertical cells in two separate windows, each controlled by node 2.

4. Construct a "central pattern generator" by connecting five neurons in a ring fashion with each neuron making an excitatory synapse onto its clockwise neighbor. Place each neuron on its own node and set the axonal delay to 5 *msec*. Investigate the use of the lookahead feature to speed up the simulation. Optional: use the performance monitoring capabilities described in the PGENESIS documentation.

5. Using the neuron model and real spike data file supplied with the PGENESIS distribution in *Scripts/experiment*, modify the parameter search of Sec. 21.7 to use a realistic evaluation function that employs a least-squared error method for comparing the simulated spikes to the actual data. Use it to find the values of the conductances

for the fast Na current and the delayed-rectifier K current that best match the data. Hint: create an evaluator element that accepts SPIKE messages and in the action handler (set by *addaction*); compare the simulated spike times against the spike times of the actual data.