A Multithreaded and Distributed System for the Simulation of Large Biological Neural Networks

---

Diploma Thesis by Jochen Martin Eppler
Submitted April 2006

Albert-Ludwigs-University
Freiburg

First Reviewer: Prof. Dr. Gerhard Schneider
Chair of Communication Systems, Faculty for Applied Sciences

Second Reviewer: Juniorprof. Dr. Markus Diesmann
Computational Neurophysics, Institute of Biology III

Co-Supervision: Dr. Abigail Morrison
Computational Neurophysics, Institute of Biology III

Bernstein Center for Computational Neuroscience

Contact Information

Jochen Martin Eppler

Computational Neurophysics, Institute of Biology III
Bernstein Center for Computational Neuroscience
Albert-Ludwigs-University

Hansastraße 9a
79104 Freiburg
Germany

Telephone: +49 761 203 9530
Fax: +49 761 203 9559
Email: eppler@biologie.uni-freiburg.de

Acknowledgements

This thesis would have never been possible without the help and support of many people.

I would like to thank Prof. Dr. Schneider for the opportunity to write this faculty spanning thesis. For the excellent supervision and the patience all along the way I want to thank Abigail Morrison and Markus Diesmann. I particularly appreciate the stimulating discussions in the great working atmosphere at the Hansastraße in Freiburg.

I am grateful to Hans Ekkehard Plesser, who implemented the original *express* kernel, which was the basis for my work.

I especially want to thank Marc-Oliver Gewaltig and the people at the Honda Research Institute Europe in Offenbach for the numerous debates and guidance through the dark valleys of multi-threaded programming and debugging.

Big thanks go to my parents for their care and backing that made my study possible in the first place. A broad smile goes out to my fellow students that made this episode in my life as much fun as possible ;-)

Finally, my very special thanks for her love and the mental support goes to Marion. Your endurance and strength in all the years is incredible. Thank you!

This work was carried out in the framework of DAAD Grant 313-PPP-N4-lk (to Abigail Morrison, Hans Ekkehard Plesser and Markus Diesmann) and was partially supported by BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg. Further support came from the Honda Research Institute.

Zusammenfassung

Die Simulation großer neuronaler Systeme wird immer mehr zu einem Grundpfeiler der modernen Neurobiologie. Dies geschieht aus zwei Gründen: zum einen sind Simulationen ein wichtiges Werkzeug um Fragen zu klären, die nicht durch Experimente oder analytische Methoden verfolgt werden können, zum anderen können sie Anregungen für neue Experimente liefern. Das Gehirn von Säugetieren ist eine sehr komplexe Struktur mit bis zu 10^{12} Neuronen. Jedes dieser Neuronen erhält Eingänge von ca. 10^4 Neuronen und produziert Ausgangssignale für etwa die gleiche Anzahl. Wenn realistische Verbindungswahrscheinlichkeiten angenommen werden enthalten die resultierenden Netzwerke mindestens 10^5 Neuronen und 10^9 Verbindungen (*Synapsen*). Dies entspricht etwa einem Kubikzentimeter Cortex (Braitenberg & Schüz, 1998). Da die Anatomie kortikaler Strukturen immer besser verstanden wird, können Wissenschaftler immer bessere Modelle von neuronalen Schaltkreisen in Computersimulationen erforschen. Die große Zahl von Neuronen und Verbindungen führt jedoch zu langen Simulationslaufzeiten und erfordert viel Speicher, weshalb hochspezialisierte Simulationsprogramme benötigt werden. Diese Arbeit beschreibt drei wichtige Erweiterungen für das Simulationsprogramm NEST, die in folgende Kategorien fallen:

Parallele und verteilte Simulation: NEST benutzt Multithreading für die parallele Simulation und ist deshalb auf Ein- oder Mehrprozessormaschinen beschränkt. NEST wird so erweitert, dass Simulationen auf Computer-Clustern ausgeführt werden können, während lokal weiterhin mehrere Threads benutzt werden können.

Erweitertes Verbindungskonzept: In NEST werden Verbindungen zwischen Neuronen in einem starren System gespeichert, das die Simulation synaptischer Plastizität erschwert. Ein neues System beseitigt diese Einschränkung und unterstützt außerdem die verteilte Simulation und einen verteilten Aufbau des Netzwerkes.

Performanceverbesserungen: Das Skalierungsverhalten von NEST war ungenügend. Durch verschiedene neue Erkenntnisse über geeignete Algorithmen und Datenstrukturen aus einem Vorläuferprojekt konnte die Leistungsfähigkeit erheblich verbessert werden. Das Skalierungsverhalten der neuen Implementation wird diskutiert.

Die im Rahmen der vorliegenden Arbeit erstellte Implementation bildet die Grundlage für die NEST2 Release.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

The simulation of large neuronal system has come to play a major role in the fast growing field of computational neuroscience. This has happened mainly for two reasons: first, simulations are an important tool to investigate and answer questions which are not tractable by experimental or theoretical methods and second, they can inspire new experiments. However, the mammalian brain is a very complex structure. It contains up to 10^{12} neurons, each of them receiving input from approximately 10^4 neurons and generating output for about as many. If realistic levels of connectivity are to be maintained, the resulting networks need at least 10^5 neurons with 10^9 connections (*synapses*). This corresponds to approximately one cubic millimeter of cortex (Braitenberg & Schüz, 1998). Because more and more is known about the anatomy of cortical structures, researchers are able to build detailed models of neural circuits that can be explored in computer simulations. The large number of neurons and connections in biological systems, however, results in long simulation times and high memory requirements and require sophisticated simulation applications.

Several tools for neural simulations exist, but most of them focus on the detailed morphology of individual nerve cells in order to capture their internal electrical dynamics. The detailed simulations these programs allow are expensive in terms of computation time and memory, moreover, they are not optimized for large neuronal systems and are thus only applicable for the simulation of single neurons or small neural networks. In this domain, the simulation programs Genesis (Bower & Beeman, 1997) and Neuron (Carnevale & Hines, 2006) have become the de facto standard due to their large user base.

Simulation programs that aim at models of large neural systems require simpler neuron models. Most of them use so-called point-neuron models (MacGregor, 1987), which have only one or a small number of electrical compartments (Larkum et al., 2001). But only a few simulators are targeting this kind of simulations. This thesis describes two extensions to the Neural Simulation Tool NEST that improve its flexibility and performance by adding support for distributed simulation and an extensible connection framework.

1.2 State of Research

NEST is a two-layered system, which is written in C++ (Stroustrup, 1997). The bottom layer consists of the simulation kernel, the top layer of a simulation language interpreter (*SLI*), which is the user interface to NEST. The separation of kernel and interpreter makes it possible to set up simulations in an easy way and, at the same time, run the simulations in the highly optimized kernel without the need for any (possibly slow) interpreter-kernel interaction. The name NEST is used for the simulation program as well as for the simulation kernel, a distinction is noted where necessary. An illustration of NEST's architecture and operational range is shown in figure 1.1.

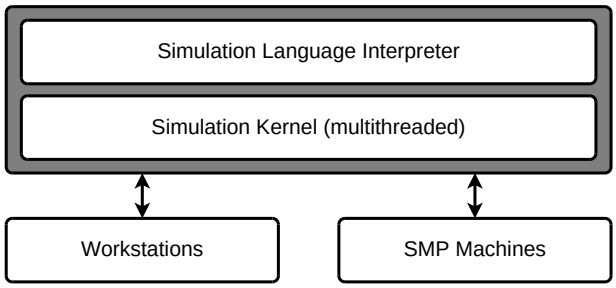


Figure 1.1: The architecture of NEST: The simulation language interpreter (top layer) offers a convenient interface to the simulation kernel (bottom layer). The program can be run on desktop and workstation computers as well as on SMP machines.

In 1999 the simulation kernel was forked and a new kernel, *Paranel* (Morrison et al., 2005), was developed to support the distributed simulation of neuronal networks on multiple computers. However, Paranel did not use the simulation language interpreter, but required the re-compilation of the program for each simulation. At the same time, a second simulation kernel, NEST, was developed to implement parallel simulation on a single computer using multithreading. The NEST kernel also uses the simulation language interpreter SLI.

1.2.1 The Simulation Language Interpreter

Using a simulation language interpreter has the advantage that simulations can be set up interactively without the need to recompile the simulation program every time a parameter is changed, or something has to be tried in an explorative way. This is important in the early phase of an experiment, where the right parameters still need to be found.

The simulation language of NEST is a simple, but complete programming language (Diesmann et al., 1995; Diesmann & Gewaltig, 2002) whose syntax is based on PostScript (Adobe Systems Inc., 1991). SLI uses post-fix notation in which the arguments in all expressions are entered before the name of the function that uses them. For example, the SLI expression for adding the two numbers 1 and 2 is '1 2 **add**'. SLI includes commands to create, manipulate and connect neural elements. It also has the usual control structures, operations, and data structures found in other programming languages. SLI has a strict type system that allows parameter checking and fine grained error handling. New commands can be implemented in C++ or in the simulation language itself.

Because of the simple syntax of the SLI language, it is easy to use the interpreter as a virtual machine for program generated code. This allows to interface the interpreter from other programs to provide online data analysis and observation of simulation results. The two interpreters can exchange commands in the respective language of the partner. Interfaces to Mathematica (Wolfram, 2003), IDL (Research Systems Inc., 1987) and Python (Lutz, 2001) exist. Moreover, current work on a MATLAB (MathWorks, 2002) interface will add a graphical user interface to NEST.

We will illustrate the key features of SLI in a small example (modified from Gewaltig & Diesmann (2006)) that simulates a neuron receiving input from an excitatory and an inhibitory neuron population. Both populations are modeled by Poisson spike generators. The simulation program tries to find an appropriate firing rate for the neurons of the inhibitory population such that the neurons of the excitatory population and the target neuron fire at the same given rate. In the first part, the simulation parameters are assigned to variables.

```
1e3    ms  /t_sim  Set
16000      /n_ex   Set
4000       /n_in   Set
5.0    Hz  /r_ex   Set
12.5   Hz  /r_in   Set
45.0   pA  /epsc   Set
-45.0  pA  /ipsc   Set
1.0    ms  /d      Set
5.0    Hz  /lower  Set
25.0   Hz  /upper  Set
0.001  Hz  /prec   Set
```

The second step creates the elements for the simulation. This is done by the SLI command `Create` that expects a neuron type and will return a handle to the new element. The Command `Set` stores the handles in variables for later reference.

```
iaf_neuron         Create /neuron  Set
poisson_generator  Create /ex_pop  Set
poisson_generator  Create /in_pop  Set
spike_detector     Create /spikes  Set
```

The third part of the simulation script configures the parameters of the spike generators. This is done by the command `SetStatus`, which expects the handle of a node and a *dictionary* as arguments. A dictionary is a set of name/value pairs, which is delimited by the symbols << and >>. By using named parameters (Finkel, 1996) it is possible to keep the class interface of the elements small and avoid *fat interfaces* (Stroustrup, 1997).

```
ex_pop << /rate r_ex n_ex mul >> SetStatus
in_pop << /rate r_in n_in mul >> SetStatus
```

The fourth part connects the two populations with the neuron, as well as the neuron with the spike detector. The command `Connect` is available in two variants. The first variant has four arguments: the handles to the pre- and post-synaptic nodes, as well as the weight and the delay of the connection. The second variant of `Connect` does not need values for weight and delay, but uses default values of 1.0 for each of them. Both versions return handles to the connections, which are discarded, using the command `pop`.

```
ex_pop neuron epsc d Connect pop
in_pop neuron ipsc d Connect pop
neuron spikes        Connect pop
```

In order to determine the optimal rate of the neurons in the inhibitory population, the network is simulated several times for different values of the inhibitory rate while measuring the rate of the target neuron. This is done until the rate of the target neuron matches the rate of the neurons in the excitatory population. The algorithm is implemented in two steps:

First, a function (`OutputRate`) is defined to measure the firing rate of the neuron.

```
/OutputRate {
   /guess Set
   in_pop << /rate guess n_in mul >> SetStatus
   spikes << /events 0 >>            SetStatus
   t_sim Simulate
   spikes GetStatus /events get t_sim div
} def
```

The function takes the firing rate of the inhibitory neurons as argument and stores it in a variable. The inhibitory Poisson spike generator is then configured accordingly. Next, the spike-counter of the spike detector (*/events*) is set to zero and the network is simulated for 1 second. The command `Simulate` takes the desired simulation time in milliseconds and evaluates the network for the given amount of time. During simulation, the spike detector counts the spikes of the target neuron and the total number can be read out after the simulation period. The return value of `OutputRate` is the mean firing rate of the neuron.

In the second step, the SLI function `FindRoot` is used to determine the optimal firing rate of the neurons of the inhibitory population.

```
{OutputRate r_ex 1.0e3 div sub} lower upper prec FindRoot
```

`FindRoot` takes four arguments: the first argument is a function whose zero crossing has to be determined. In SLI, the characters { and } define a *pure function* without a name. Pure functions can be assigned to variables or serve as arguments for other functions (Finkel, 1996). Here, the firing rate of the target neuron should equal the firing rate of the neurons of the excitatory population. The next two arguments are the lower and upper bound of the interval in which the zero crossing is sought for. The final argument is the desired precision of the zero crossing.

1.2.2 The NEST Simulation Kernel

NEST is the third generation simulation kernel. In previous versions, the elements were communicating by directly calling their member functions. This is the most general way of interaction, however, it makes parallel processing impossible, since every element has access to the data members of all other elements at any time. In NEST the elements communicate by exchanging events (see section 3.2) which encapsulate and transport data. This flexible way of communication will be covered in more detail in chapter 5. NEST has a generic element concept, where the network elements, including the devices, are derived from a common base class. This base class defines the interface to create, configure, and connect elements section 3.1. A special class of elements are sub-networks that group elements of a network. Thus it is possible to divide a large system into smaller pieces to model parts like cortical areas.

Each network element stores a list of all elements it is connected to, along with the parameters for each connection. In particular, the weight of the connection (i.e. amplitude of the post-synaptic potential) and the transmission delay (see chapter 4). This static representation of the connectivity, however, renders simulations of learning and plasticity very difficult, because it is not easily possible to change the connection weight by a learning algorithm that depends on the activity of both connected neurons.

To provide a convenient user interface to the simulation infrastructure, NEST is integrated with the simulation language interpreter, which has been explained in section 1.2.1.

The NEST kernel supports parallel network simulation by using multiple threads on SMP[1] and desktop computers. The program is executed in multiple *threads* (Lewis & Berg, 1997) that all have full access to the data of the program. On SMP machines several threads can be executed in parallel and solve a common task. They do not require a special communication infrastructure, as each of them has access to the same variables. Thus, in principle, multithreading allows near optimal use of a computer's processing power and memory. But multi-threaded programming is difficult and can lead to severe performance problems (see section 1.2.4).

1.2.3 The Paranel Simulation Kernel

The Paranel simulation kernel is a branch of the first generation simulation kernel and was developed to investigate how simulations can be distributed over networked computers. While multi-threaded simulations only run in one process on a single computer, distributed simulations run in several processes that may be distributed to multiple computers. An interesting advantage of distributed simulation is that each computer not only contributes its computing power, but also its memory. This was the main motivation to implement the Paranel kernel. Thus, Paranel can solve problems that would exceed the memory of a single computer. But, since different parts of the simulation run in different processes, Paranel must solve two problems: first, it must divide the simulation into equal chunks and assign them to the processes. Second, each process needs to send its spike data to the neurons on remote processes.

The first problem is solved by assigning the neurons to the processes by using a simple modulo operation. This leads to an equal distribution of the elements. The second problem is solved by using an external library that passes messages between processes in a network of computers. There are a number of different libraries available for this purpose. Paranel uses the Message Passing Interface (*MPI*, Message Passing Interface Forum, 1994).

Because Paranel is derived directly from the first generation of the simulation kernel (Diesmann et al., 1995), it still suffers from problems that are already solved in later versions. Some of these problems are:

- Neurons and devices do not share a common interface or base class.
- It is not possible to structure networks or build hierarchical networks.
- Interaction between neurons is restricted to spikes.
- The user cannot easily inspect or modify the parameters of a neuron, device, or neuronal connection.

[1] The term SMP will refer to all shared memory multiprocessor architectures, see Tanenbaum (1999)

Paranel, however, features a flexible system to connect neurons. Different synapse prototypes are available that can be used to model learning, e.g. synaptic short term dynamics (e.g. Tsodyks et al., 1998) or spike-timing dependent plasticity (Morrison et al., 2005). Moreover, Paranel can compress the connection information in several ways to reduce the memory requirements. Paranel does not use the simulation language interpreter SLI for network setup and administration. Thus, the simulation program has to be re-compiled every time a parameter has to be changed.

1.2.4 Comparing NEST and Paranel

One can can directly compare the absolute simulation time and the scaling of Paranel and NEST, by simulating the same network with both simulation kernels. The results of this comparison are shown in figure 1.2. Part (A) shows the absolute run times of NEST (solid line) and Paranel (dashed line) in log-log representation. The gray line indicates linear scaling, meaning that the run time T_n with n processors is given by $T_n = \frac{T_1}{n}$. Part (B) shows the speedup of NEST (solid line) and Paranel (dashed line) in log-log representation. The speedup s_n with n processors is defined as $s_n = \frac{T_1}{T_n}$ (see chapter 6). The gray line again indicates linear speedup. In both panels it is obvious that Paranel has better scaling than NEST: if executed on 8 processors, Paranel has a speedup of about 16. In contrast, NEST's scaling is not even linear and seems to saturate with more than 2 processors.

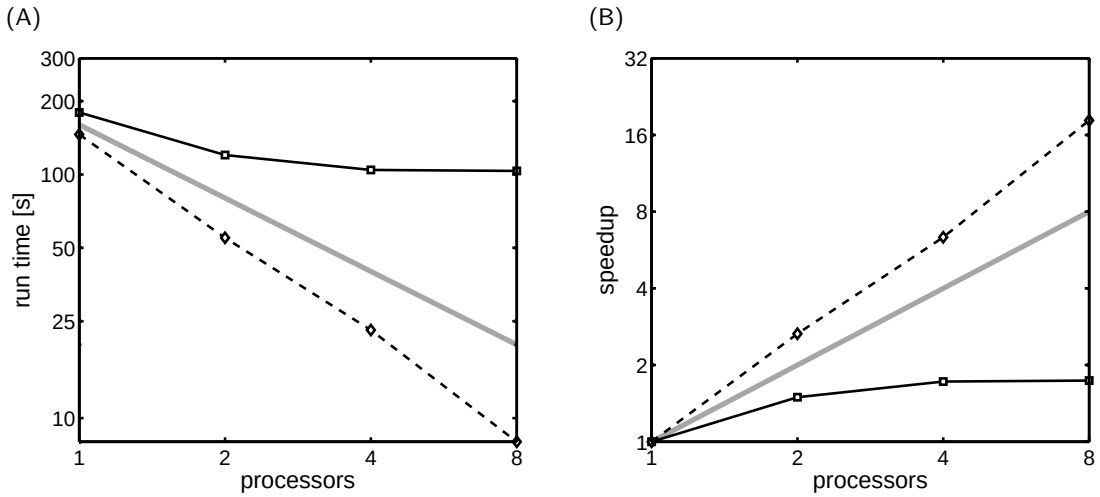


Figure 1.2: Scalability of NEST and Paranel with respect to number of processors: The network contained 10^4 neurons, with 1000 random connections each; the exact model is described in Brunel (2000). Solid line: NEST; dashed line: Paranel. The gray line indicates linear speedup. (A) Simulation time against number of processors, log-log representation. (B) Corresponding speedup (see equation 6.1) against number of processors, log-log representation.

What are the reasons for the large performance difference between NEST and Paranel? The main reason is the different memory layout of the two kernels: In Paranel, each process has its own region of memory and is assigned to one processor. NEST uses multiple processors

within a single process. This means that all threads use the same region of memory. As modern processors compute much faster than the memory can deliver data, each CPU contains a *cache* as fast working memory. If two processors operate in different regions of the memory, they can optimally use their cache. This is the case for Paranel. And since each processor also adds cache memory, Paranel scales better than linear. However, if two processors work in the same region of memory, one of them has to reload its cache from the slow memory. This is called *cache thrashing* and is a common problem of multithreaded applications. Because NEST stores all network nodes in a contiguous piece of memory, it is very likely that two or more threads will operate on the same memory region. This causes the other processors to reload their cache. Since the probability for cache thrashing increases with each additional thread, this explains the bad scaling of NEST (see section 2.2.5).

For the simulation of neuronal systems, the speed of memory access is more critical than the processing speed. This is because the elements are individually cheap to update but highly interconnected. In Paranel, the number of memory accesses is reduced by keeping the data in the processor's cache as long as possible. The exact algorithm allowing this is described in section 5.5.

Another problem of NEST is its static connection representation, in which the synapses are stored distributed across all elements, each of them containing its own target list. This representation does not support plasticity and learning synapses. The connection system in Paranel, however, is more flexible and supports different synapse types, which can be used to implement heterogeneous networks.

Multithreaded programming requires thread safe data structures which can be critical to the performance of a program. However, this way of programming provides great flexibility, because class objects can call each other's member functions directly. This is not the case with distributed simulation, where each exchange of data between the processes requires special treatment, i.e. packing by the sender, sending, and unpacking by the receiver.

1.3 Task Definition

My task for this thesis is to integrate the two simulation kernels NEST and Paranel into a new simulation kernel. Instead of developing a new kernel from scratch, we start with the NEST kernel, which already contains many of the desired features. Features that are superior in Paranel or missing in NEST, will be added to the new kernel. What follows, is a list of desired properties for the new kernel.

1.3.1 Multithreaded and Distributed Simulation

Multithreading is a flexible technique for parallel computation on SMP machines. It offers fast interaction between the elements ranging from simple spikes to the exchange of arbitrarily complex data. For this reason we will further develop this approach, although its performance is not optimal at the moment. To allow simulations that are larger than the memory available on a single computer, we need distributed simulation. This has the additional advantage that the network construction can be parallelized.

The first step is to port the communication facilities from Paranel to the NEST kernel. For this, we must modify the representation of the elements and their connections. Moreover,

the scheduling algorithms and data structures have to be adapted.

1.3.2 Connection Management

The connection framework of Paranel has some distinct advantages over NEST. It supports different synapse types to build heterogeneous networks, learning, and plasticity. Because of the differences in the representation of elements, it is impossible to port Paranel's system directly. My objective is to implement a new connection framework for NEST with all the features of Paranel's connection system.

1.3.3 Interpreter Integration

The simulation language interpreter is a convenient interface user interface and spares the user from re-compiling the program after parameters changes. Two innovations in the new kernel require modifications and extensions to SLI:

1. The new connection framework requires commands for SLI to manipulate synapses and defaults for the synapse types.
2. We also want to use the interpreter for distributed simulations. This is a problem, since the interpreter is based on a simple stack machine (Aho et al., 1988), which is inherently serial. The easiest solution is not to use the interpreter interactively, but to use one interpreter per process and have the kernel functions pick the relevant commands from a script. An interactive mode for the interpreter is described in chapter 7.

1.3.4 Reproducibility

Reproducibility means that a program will yield the same result if it is run several times with the same parameters. This is the normal behavior for uniprocessor computers. On computers with more than one processor, however, this may no longer be true, since the order in which different threads are executed is not deterministic. It is an important requirement for the new simulation kernel that it produces the same results when it simulates with multiple threads or in many distributed processes. To be more precise, it should yield the same results for any simulation, distributed or multi-threaded, where the number of virtual processes (number of processes $\times$ number of threads) is kept constant.

1.3.5 Performance Improvements

We want the new kernel to scale as well as Paranel. This can be achieved by using the algorithms and data structures from Paranel for both, multithreaded and distributed simulation. This will reduce NEST's cache problems by separating the memory of each thread.

1.4 Layout of the Thesis

In chapter 2, the data structures and algorithms for the network setup are introduced. This includes the elements as well as the connections between them.

Chapter 3 explains the construction of elements, their functionality, and their member variables. It also introduces The events that are used during connection setup and for the interaction of elements during simulation.

Chapter 4 explains the new framework for connection establishment and event transmission. The last section gives some examples for the SLI interface to the new connection subsystem.

Chapter 5 introduces the algorithms and data structures necessary for the scheduling of the simulation. The event delivery mechanisms are explained together with the communication between processes.

In chapter 6, the benchmark results are presented together with a theoretical analysis of the scaling behavior. The benchmarks compare the absolute run time and scalability of NEST, NEST2 and Paranel.

Finally, chapter 7 contains a discussion of the solutions that were developed in this thesis and compares them to alternative approaches together with ideas for possible future developments.

Chapter 2

Network Representation

The simulation kernel has two main classes: the network class, which stores the network elements and their connections, and a scheduler class, which is responsible for updating the network and which will be explained in chapter 5.

A neural network can be seen as a directed graph, in which the vertices denote the neural elements (*nodes*) and the edges represent the connections between them. The class diagram in figure 2.1 shows the most important data members and functions of class `Network`.

Network

`models: vector<Model*>`
`nodes: vector<Node*>`
`scheduler: Scheduler`

`add_node(): void`
`get_node(id:int, thread:int): void`
`template<class NodeT> register_model(name:string): void`
`send(event:Event&, lag:int): void`
`send_to_node(event:Event&, lag:int): void`

Figure 2.1: UML diagram for class `Network`: The most important member variables (top part) and functions (bottom part) of class `Network`.

The elements of the network are created by factory objects, so-called *models* (see section 3.1.4). Models are added to the network's *models* list during initialization by calling `register_model()`. A model's position in the list is called its *model id*. Nodes are created by using the SLI command `Create`, which expects the *model id* as argument. `Create` then calls the C++ function `add_node()`, which creates a new node from its model and stores the node in the *nodes* list. The two functions `send()` and `send_to_node()` are used for event delivery during simulation and will be explained in section 5.8 and section 3.2, respectively.

2.1 The Network in NEST

In NEST, a network is represented as the directed graph $\mathcal{G}$, defined as $\mathcal{G} = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ the set of directed edges. For storing the network on a

computer, the graph can be represented as adjacency matrix (Gross & Yellen, 1999). A sparse matrix can be represented more compactly as adjacency list that stores the target lists for each node in the network. Figure 2.2 shows an example network in this representation.

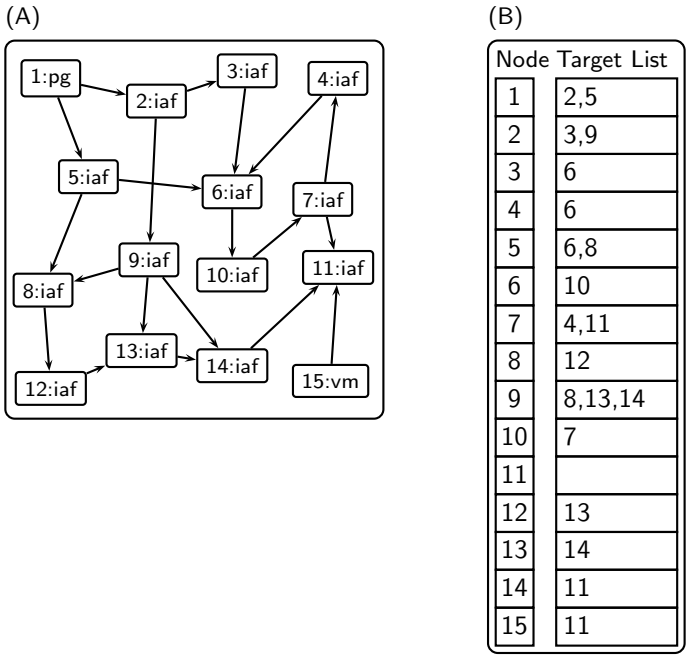


Figure 2.2: Network representation as adjacency list: A small example of a neural network consisting of 15 nodes: 1 Poisson spike generator (pg), 13 integrate-and-fire neurons (iaf), and 1 voltmeter (vm). (A) The network as a directed graph. The edges of the graph contain a node's *global id* and its *type*, separated by a colon. (B) The corresponding adjacency lists. Each node is assigned a list of its targets.

In the original NEST kernel, the network is represented by an adjacency list that is stored distributed in the target lists of each node. This network representation has the following problems that complicate the implementation of the new features:

1. It is not possible to distribute the simulation, which means that the network size being limited by the amount of local memory.
2. The network is is constructed by only one thread. This leaves the other threads idle and may take a lot of time for large networks.
3. The thread that updates a node will also write this node's events to the input buffers of all targets, which may belong to another thread. This causes two problems: first, cache thrashing occurs whenever the input buffers of a node are written. Second, we need to prevent that two threads access the same data simultaneously. Both are severe performance bottlenecks especially for generator devices (e.g. random spike generators, see section 3.1.1) that deliver large amounts of data.
4. The connection information is hidden in the nodes. Thus, it is difficult to access the con-

nection parameters for performing learning and plasticity algorithms (see section 1.2.4).

2.2 The Network in NEST2

To achieve a representation of the network that is suited for both distributed and multithreaded simulation, the network is split into several chunks that are assigned to threads or processes. Every node is then assigned to exactly one chunk.

In order to gain more flexibility for the connections, we explicitly store them in the kernel, separated from the nodes. A new object, the `ConnectionManager`, connects nodes, administrates connection information, and delivers events during simulation. In the following section, we will only describe the new network representation. The details of the new connection framework are the topic for of chapter 4.

2.2.1 Virtual Processes

The network is split into several parts by introducing *virtual processes*. Each virtual process is responsible for one part of the network. Virtual processes are independent of the actual number of threads and processes and combine the different network representations for multithreaded and distributed simulation in one abstract description. Each virtual process stores its nodes in a vector<Node*> that is referred to as its *node list*.

2.2.2 Assigning Nodes to Virtual Processes

The fact that each node is updated by a single thread introduces a performance bottleneck for devices (see section 2.1). This bottleneck is removed in NEST2 by replicating the device nodes once for each virtual process. The connection algorithm in section 4.2 is built so that connections of neurons from and to devices are established always with the device on the neuron's virtual process. This way, from the view of an individual node, all connections appear to be on the same virtual process, which effectively splits the computational load over all virtual processes. To keep the creation time and memory overhead minimal, the devices are created as children of sub-networks, which provide efficient construction by using the memory allocator (see section 3.1.3).

In contrast to the devices, the neuron nodes are created only once. Each virtual process has an unique id. To distribute the nodes evenly across the virtual processes, each node is assigned to exactly one of them. The id id_v of the virtual process a node belongs to is given by $id_v = id_n \mod T$, where id_n is the global id of the node and T the number of overall virtual processes. This distribution algorithm is the best approach for the general case, where nothing is assumed about the activity in the network. For structured networks where the communication patterns are known in advance, a better strategy may always exist. This problem is also addressed in the discussion in chapter 7.

2.2.3 Accessing Nodes in Virtual Processes

The modulo algorithm used for the distribution of the neurons has consequences for way the nodes are addressed inside the virtual process they are assigned to: If no precautions are taken, the nodes are stored in the node lists in order of creation, as seen figure 2.3. This requires a

lookup table in each virtual process to map the *global id* of a node to its index position in the *nodes* list. This table has to be used for every node access and adds an indirection for each access.

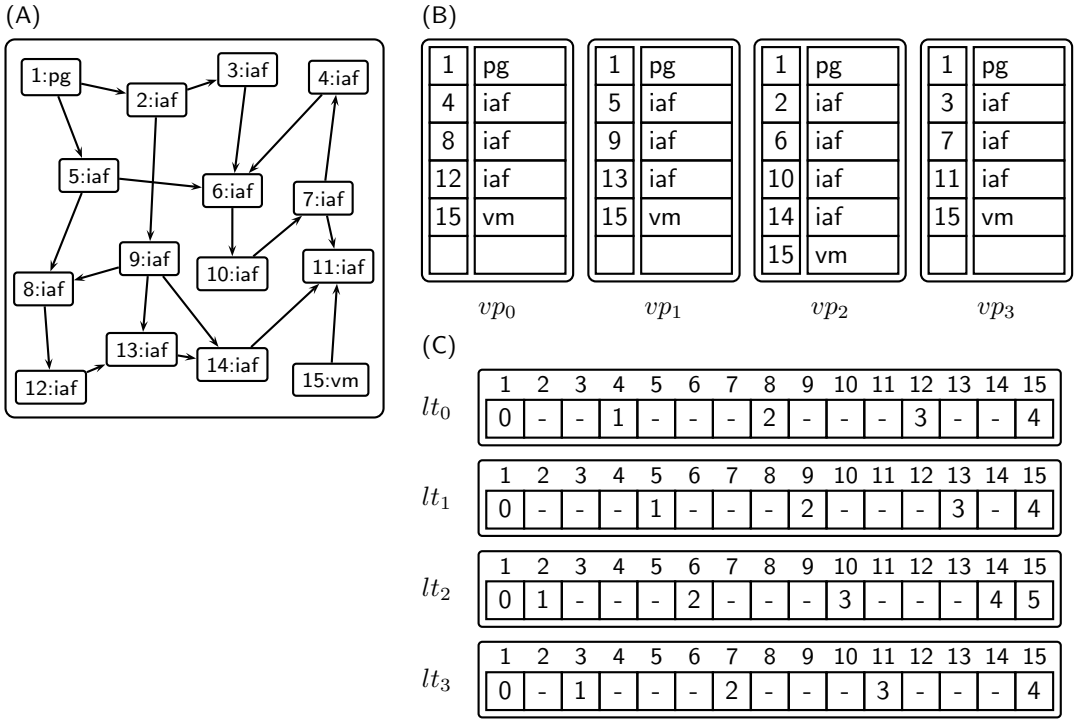
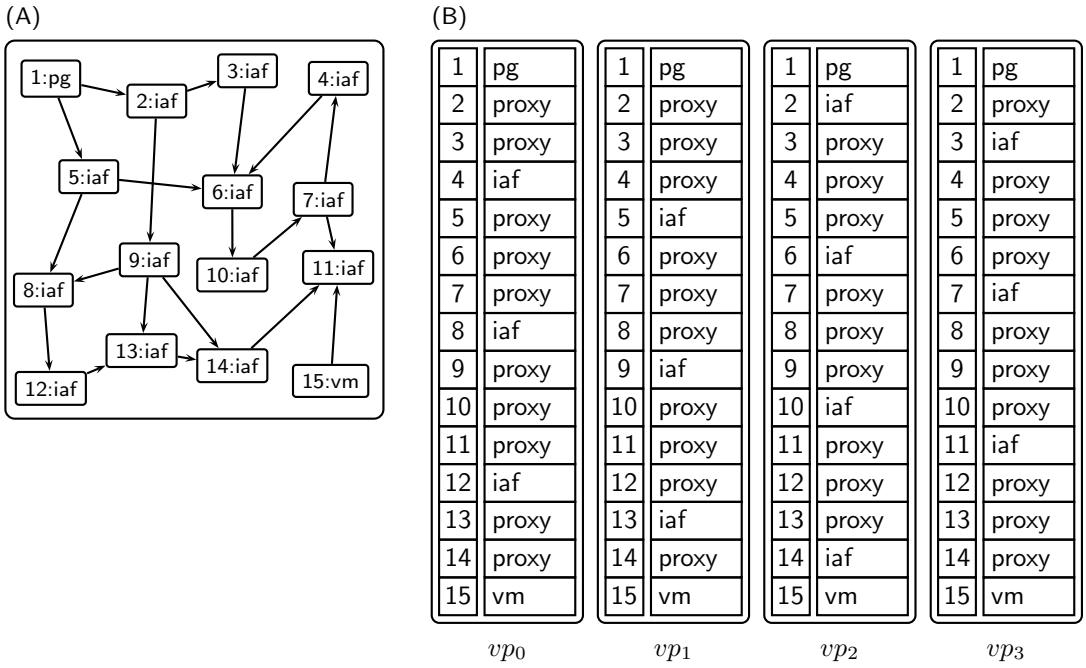


Figure 2.3: Network representation on virtual processes using lookup tables: Same network as in figure 2.2. (A) The network as a directed graph. (B) The network representation using virtual processes ($vp_0, \ldots, vp_3$). The nodes are stored in the *nodes* lists of the virtual processes in the order of their creation. The first column shows the node's *global id*, the second column shows its type. (C) Lookup tables for the mapping between *global id* and local index position. The nth entry of table lt_i contains the index of the node with *global id* n in the *nodes* list of virtual process vp_i.

Another possibility to solve the addressing problem is to create local *proxy nodes* for all nodes that reside in other virtual processes. This does not require a table nor any pre-calculations for accessing the nodes, but uses the global id of a node as index into the node list directly. This leads to better performance during connection setup and simulation where node access id most frequent. Proxy nodes offer the additional advantage that they can be used to hold additional information about the nodes they represent. In the current implementation, the proxy nodes store the *model id* of the real node, thus making it possible to obtain a complete printout of the network by just looking at one virtual process. The proxy nodes are built as lightweight derivatives of class `Node`, but nonetheless may account for a significant portion of the memory needs for the nodes for large numbers of virtual processes. If this approach becomes too expensive, the performance penalty of the lookup table may again be a reasonable alternative (see chapter 7). A representation of the example network onto four

virtual processes that uses proxy nodes is shown in figure 2.4



	vp_0	vp_1	vp_2	vp_3
1	pg	pg	pg	pg
2	proxy	proxy	iaf	proxy
3	proxy	proxy	proxy	iaf
4	iaf	proxy	proxy	proxy
5	proxy	iaf	proxy	proxy
6	proxy	proxy	iaf	proxy
7	proxy	proxy	proxy	iaf
8	iaf	proxy	proxy	proxy
9	proxy	iaf	proxy	proxy
10	proxy	proxy	iaf	proxy
11	proxy	proxy	proxy	iaf
12	iaf	proxy	proxy	proxy
13	proxy	iaf	proxy	proxy
14	proxy	proxy	iaf	proxy
15	vm	vm	vm	vm

Figure 2.4: Network representation on virtual processes using proxy nodes: Same network as in figure 2.2. (A) The network as a directed graph. (B) The network representation using virtual processes $(vp_0, \ldots, vp_3)$. Nodes that are on remote virtual process have local proxies. The first column shows the node's *global id*, the second column shows its type.

2.2.4 Random Number Generation and Reproducibility

In the simulation of neuronal systems, random numbers are needed to set up random connections (see section 4.3.1) or to provide stochastic background activity to a network (see section 1.2.1). An important property of NEST2 is that a neuronal simulation with an arbitrary but fixed number of virtual processes will yield the same results, regardless of the way (i.e. multithreaded or distributed) it is simulated. To achieve this, each virtual process has its own random number generator (RNG) with its own seed. Because a node is always assigned to the same virtual process, it will always access the same RNG. But this is still insufficient to achieve reproducibility. Additionally, the random numbers must always be drawn in the same order. This is guaranteed by the connection algorithm that is described in chapter 4.

2.2.5 Multithreaded Representation

On computers with several processors, multithreading allows the network update to be performed in parallel. But we must guarantee that no two threads can write to the same memory address at the same time. This is usually called *thread safety* and can be achieved in two ways:

1. Algorithmically: The programmer can use barriers, special data structures like mutual exclusion locks (*mutex*, see Lewis & Berg (1997))) or semaphores, to protect critical regions of code. If one thread enters the critical region, it closes the barrier and the remaining threads must wait until the barrier is re-opened by the first thread at the end of the region.
2. By the data structure: Since it is safe to simultaneously read from the same memory location, a data structure is thread safe, it is has separate fields for each thread. The ring buffers of NEST (see section 3.1.2) are designed by this principle.

We have already mentioned that cache thrashing is a problem in multithreaded programming. When a process is requesting a certain datum from memory, the processor will first check its cache for the datum d. There are two possibilities:

1. $d \notin$ cache (*cache miss*): fetch the datum from main memory into the cache and proceed.
2. $d \in$ cache (*cache hit*): proceed.

The processor cache has an access time of approximately 1 ns, which is about 10 times faster than the main memory. This means that a program is executed 10 times faster if it runs completely in the cache and no main memory access is needed. Moreover, the cache is usually large compared to the amount of memory requested, so most processors copy not only the requested datum, but also a surrounding range (a *cache line*) to their caches to decrease the probability of a cache miss. In this situation the layout of main memory influences the effectiveness of the cache strongly, or more precise: it is best, if the data for a processor is arranged sequentially in the main memory. This is illustrated in figure 2.5 for two different possible memory layouts.

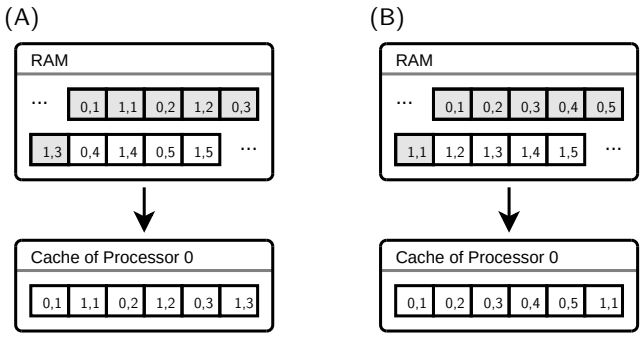
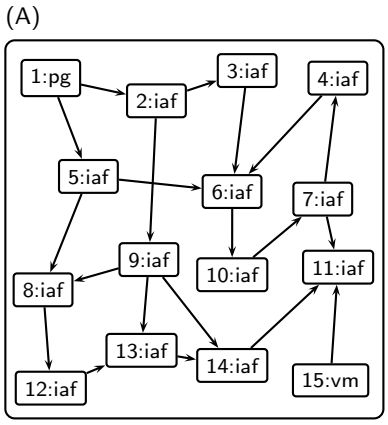


Figure 2.5: Comparison of memory layout with respect to cache utilization: Block i, j is requested by thread i in the jth access, grey blocks indicate the capacity of the cache. (A) Alternating memory layout: the first cache miss of processor 0 occurs after three successful attempts. (B) Ordered memory layout: no cache miss for the first five accesses.

In Paranel, the network is distributed on multiple processes and the memory of the different processes is separate by definition. As every process is executed by one processor, cache problems are unlikely to occur. NEST2 uses threads and distributed processes, which makes it more difficult to appropriately separate the memory regions for each processor. The order in

which network elements are created decides where they will be in memory. A solution to this problem is the separation of nodes by thread that is performed by the pool allocator of the models (see section 3.1.3).

The large number of proxy nodes used in figure 2.4 is not necessary for a purely multithreaded simulation setup, because all nodes are in the same process and can be accessed directly. Moreover, we can store all the nodes of all virtual processes in a single node list. The multithreaded representation of the example network is shown in figure 2.6.



(B)

1	-	pg
2	2	iaf
3	3	iaf
4	0	iaf
5	1	iaf
6	2	iaf
7	3	iaf
8	0	iaf
9	1	iaf
10	2	iaf
11	3	iaf
12	0	iaf
13	1	iaf
14	2	iaf
15	-	vm

$$\bigcup_{i=0,\ldots,3} vp_i$$

Figure 2.6: Multithreaded network representation: Same network as in figure 2.2. (A) The network as a directed graph. (B) The network represented for multithreaded simulation. The virtual processes are collapsed into a single node list, denoted by the union of vp_i, with $i = 0, \ldots, 3$. The first column shows the node's *global id*, the third column shows its type. The second column contains the number of the thread a node is assigned to; a '–' indicates that a node is created for each thread.

2.2.6 Distributed Representation

If a neuronal simulation is distributed onto multiple processes, the processes have direct access only to a part of the network and proxy nodes are used as local placeholders for the remote nodes. Each process can be run either with a single or with multiple threads. Because the connections are stored separately from the nodes, the distributed network representation is just an extension of the multithreaded representation, which has been explained above. For the example network, the distributed network representation is illustrated in figure 2.7, where two processes with two threads each are used.

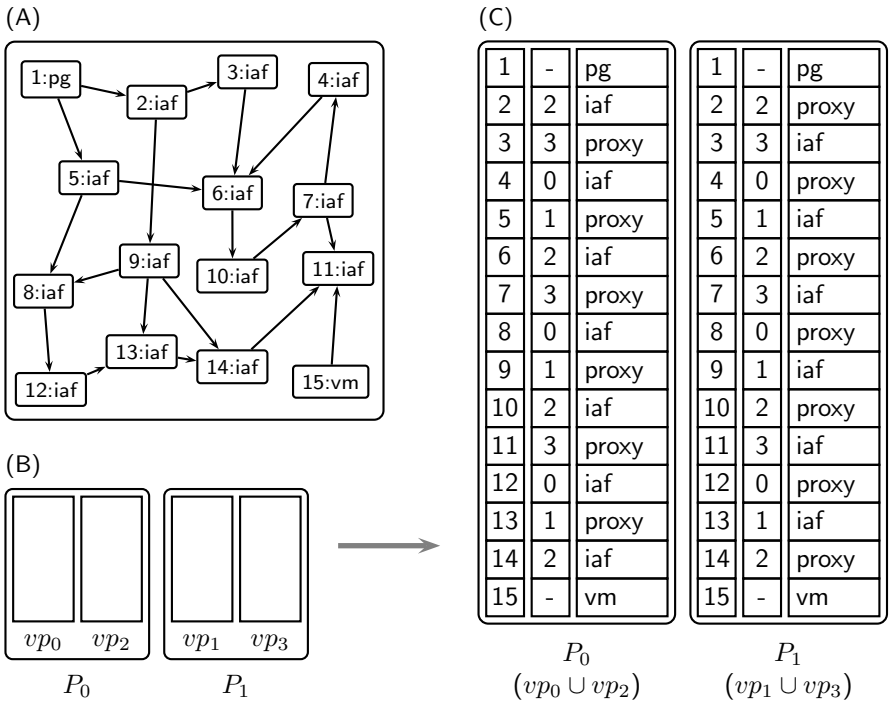


Figure 2.7: Distributed and multithreaded network representation: Same network as in figure 2.2. (A) The network as a directed graph. (B) A Sketch of the distribution of the four virtual processes onto two processes, P_0 and P_1. (C) The node lists of the two processes contain the nodes of two virtual processes each, denoted by $vp_0 \cup vp_2$ and $vp_1 \cup vp_3$ respectively. The first column shows the node's *global id*, the third column shows its type. The second column contains the number of the thread a node is assigned to; a '–' indicates that a node is created for each thread.

Chapter 3

Network Elements and their Interaction

3.1 Nodes

The network (see chapter 2) is built step by step from single elements, called *nodes*. Class `Node` is the common base class for them and is sufficient to implement a wide range of neuron models, devices and auxiliary elements. Its prototype is shown in the class diagram in figure 3.1. In order to keep the interface as general as possible, the base class does not rely on any information about the internal processes of a node.

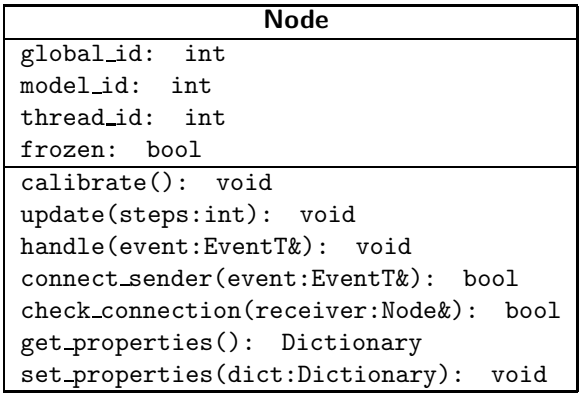


Figure 3.1: UML diagram for class `Node`: The most important member variables (top part) and functions (bottom part) of class `Node`. `EventT` represents an event of arbitrary type that depends on the actual node.

3.1.1 Node Types

Different node types are derived from the base class `Node` as shown in the inheritance diagram in figure 3.2). The derived nodes are either atoms or compound elements. The compounds, mainly sub-networks, are used to build structured models of cortical circuits and hierarchies.

They are excluded from the update of the network during simulation, as they simply act as containers for the atoms, which may either be neuron models or devices and participate actively in the simulation.

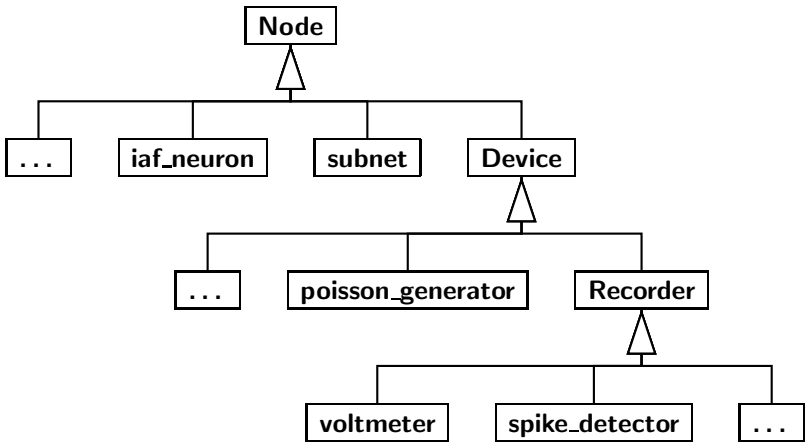


Figure 3.2: Class hierarchy for nodes: neurons and sub-networks are derived directly from the base class, whereas devices and recorders have separate base classes that add additional functionality.

In the following we will describe the most important member variables and functions of base class `Node` (see figure 3.1) that are common for all node types:

global id: Within the network, each node is identified by a unique number, which is called *global id*. It is assigned by class `Network` at the time of the node's creation.

model id: The *model id* is a reference to the model object that created the node. Models are factory objects for nodes and will be explained in section 3.1.4.

thread id: Each node is assigned to exactly one thread. The thread number is stored in the node's *thread id*.

frozen: In order to exclude parts of the network from the updating process, nodes can be *frozen*, e.g. sub-networks and proxy nodes are always frozen.

All functions of class `Node` are virtual and can thus be overloaded by the derived node types to implement custom behavior. This is shown exemplarily for a typical neuron model, the *integrate-and-fire* (I&F) neuron (see listing 3.1). Pure virtual functions are marked with a star.

`calibrate()`*: In this function, the node can adjust to the global network parameters before a simulation (see section 5.4). The I&F neuron only adjusts its internal buffers in `calibrate()`.

`update()`*: During simulation, each thread iterates over its nodes and calls their `update()` function (see chapter 5). It computes the new state of the node and, if necessary, sends events to other nodes. The function receives the number of time steps it has to advance

its state as argument (*steps*). Although the events are constrained to a fixed time grid, a node can, in principle, use any resolution to propagate its state (see section 5.5). The `update()` function of the I&F neuron is shown in lines 6 through 14 of listing 3.1, where a `SpikeEvent` is created and sent, if the membrane potential has crossed the threshold. The update is carried out in *steps* steps.

`handle()`: This function processes incoming events during simulation. It has to be implemented for each event type a node can receive. The I&F neuron may receive `SpikeEvents` (lines 13 to 16), `CurrentEvents` (lines 23 to 26), and `PotentialRequests` (lines 33 to 40). The latter are directly answered by sending a `PotentialEvent` (lines 35 to 39) to the requesting node.

`connect_sender()`: This function is called on the receiver during the connection type check. The default implementation of the base class returns *false* for all event types. For each event type that a derived node can handle, `connect_sender()` must be implemented to return *true*. For the I&F neuron they are shown in lines 18-21, 28-31 and 42-45 of the listing.

`check_connection()`: This function is called on the sender to check the compatibility of sender and receiver before a new connection is registered (see section 4.2). In the listing of the I&F neuron, the connection check is implemented in lines 47 to 53. The function calls the corresponding `connect_sender()` function on the receiver, which will return *true* only the derived node has an implementation for the corresponding event type. A detailed description of the algorithm for the compatibility check is given in section 4.2.1.

`get_properties()`* and `set_properties()`*: These functions are used to retrieve and set parameters of a node. They correspond to the SLI functions `GetStatus` and `SetStatus`, which have been introduced in the example in section 1.2.1.

```
1  void iaf_neuron::update(int steps)
2  {
3    for (int lag = 0; lag < steps; ++lag)
4    {
5      // update the internal state
6
7      if (mem_pot >= threshold)
8      {
9        SpikeEvent e;
10       e.set_sender(*this);
11       network.send(e, lag);
12     }
13   }
14 }
15
16 void iaf_neuron::handle(SpikeEvent& event)
17 {
18   // buffer the incoming event for later processing
19 }
20
21 bool connect_sender(SpikeEvent&)
```

```
22 {
23   return true;
24 }
25
26 void iaf_neuron::handle(CurrentEvent& event)
27 {
28   // buffer the incoming event for later processing
29 }
30
31 bool connect_sender(CurrentEvent&)
32 {
33   return true;
34 }
35
36 void iaf_neuron::handle(PotentialRequest& request)
37 {
38   PotentialEvent event;
39   event.set_sender(*this);
40   event.set_receiver(request.get_sender());
41   event.set_potential(mem_pot);
42   network.send_to_node(event);
43 }
44
45 bool connect_sender(PotentialRequest&)
46 {
47  return true;
48 }
49
50 bool iaf_neuron::check_connection(Node& r)
51 {
52   SpikeEvent e;
53   e.set_sender(*this);
54   e.set_receiver(r);
55   return r.connect_sender(e);
56 }
```

Listing 3.1: Excerpt from the implementation of the integrate-and-fire neuron.

Neuron models

Different neuron models describe different types of neuronal dynamics. Important examples are the *integrate-and-fire* neuron model (Tuckwell, 1988) and the Hodgkin-Huxley neuron model (Hodgkin & Huxley, 1952). In NEST, each model is implemented as a distinct node type. The neuron models can be used together with devices to build the desired neuronal systems. Currently, all implemented neuron models are *point neurons*.

Devices

In every neurophysiological experiment there are devices that generate stimuli, measure, and record neural activity. In NEST, device nodes are derived from a common base class, `Device`, which is derived from the top-level base class `Node`. There are two groups of devices: generators generate input to the neurons (e.g. currents or spikes) and recorders, which record different aspects of a neuron (e.g. membrane potential or spike output) and can be used for observation and visualization of the network activity. The `Device` base class adds a timer to the nodes, which controls the start and stop of the device. class `Recorder` adds a mechanism for file handling to the `Device` class that is used by recording devices like `voltmeter` and `spike_detector`.

3.1.2 Ring Buffers

During simulation, most nodes receive events that they have to process. Each event is sent via a connection that has a certain transmission delay. Thus it is necessary to buffer the event until it is due. Unlike other systems, NEST has no central event queue, but the events are queued at the receiver's side in *ring buffers*. The ring buffers are thread safe, because they provide separate rings for write accesses by different threads (see figure 3.3 (A)). Although the buffer is written by all threads in parallel, it is read by only one thread when the node is updated, Because the current ring buffer segments need to be reset after readout, which is a write, the thread that updates the node and resets the buffer will inevitably invalidates the caches of all other threads (processors). This effect contributed significantly to the bad performance of NEST.

In NEST2, each node is assigned to exactly one thread. The connection system (see chapter 4) ensures that all connections are set up so that each node only has targets that are on the same thread, which means that the ring buffers of a node are always read and written by the same thread. Thus, NEST2 uses a simple ring buffer (see figure 3.3 (B)), which occupies less memory and does not interfere with other threads. The two different versions of the ring buffer are illustrated in figure 3.3.

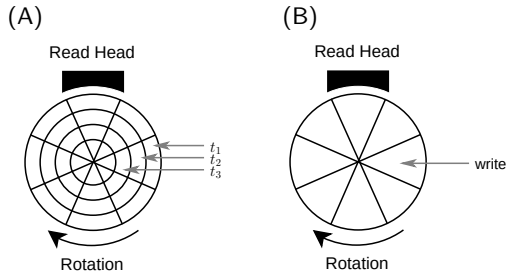


Figure 3.3: Illustration of thread safe and non thread safe ring buffers: For readout of all fields, the buffer rotates under the read head. (A) Thread safe version. Each segment of a slice is assigned for writing to a single thread. Arrows depict write access, the t_i represent threads. (B) Non thread safe version. The algorithm has to guarantee thread safe write access.

3.1.3 Memory Management

The standard allocator in C++ is suited for general needs, but involves some overhead, both in space and time, that can be avoided by using a custom allocator. The allocator of NEST2 is optimized for efficient creation and destruction of many small objects of approximately the same size. It manages a separate memory pool for each thread and new objects are automatically allocated in the corresponding memory segment. This facilitates efficient multithreaded simulation (see section 2.2.5), because collisions are avoided. Figure 3.4 illustrates the memory allocator of C++ , NEST, and NEST2.

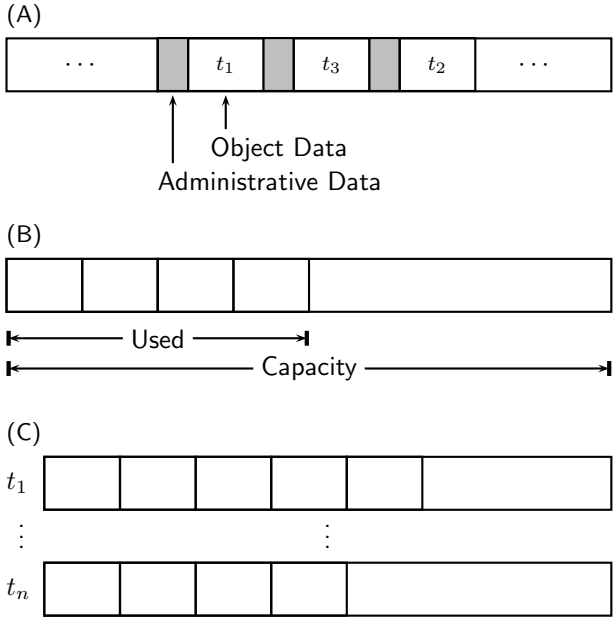


Figure 3.4: Memory allocation using different allocators: (A) The C++ allocator creates new objects at an arbitrary location in memory in the order of their creation (indicated by the t_is). Each object is preceded by a block of administrative data. (B) The pool allocator of NEST. A large block of memory is obtained from the operating system. Objects of identical type are are created in this block and do not require additional data. (C) In NEST2 each thread $(t_1, \ldots, t_n)$ has its own pool allocator.

3.1.4 Node Construction

Nodes are created by factory classes (Gamma et al., 1994) that are derived from a common base class `Model`. Each type of neuron, device, or sub-network has its own `Model` class. The models are implemented as C++ template classes in order to make it as easy as possible to introduce new node types to NEST. For example, to register the I&F neuron model with the kernel, the command 'network. register_model <iaf_neuron>("iaf_neuron");' is sufficient. The type of the model is given as template parameter in angle brackets, the argument in parenthesis is its name. Models have to be registered with the simulation kernel during initialization. The

kernel assigns a unique id to each model. In addition to construction, models are used to administrate default values for the different types of nodes. The model dictionary of SLI maps the numeric model identifier to descriptive names.

3.2 Events

Biological neurons interact mainly by exchanging spikes, which are discrete all-or-nothing events. Respectively, a `SpikeEvent` does not contain any information other than the time of occurrence. For performance reasons, the sender may set the *multiplicity* of a spike to send multiple of them within a single event. The devices are using special event types depending on their task. A current generator is sending `CurrentEvents` to its targets, which then will be processed in the next update. Another scheme is used by the voltmeter, which is sending a `PotentialRequest` that is answered immediately with a `PotentialEvent`. For this, the function `Network::send_to_node()` is used, which delivers an event directly to a single node. Further event types exist that enable the flexible exchange of arbitrary data by the nodes.

Typed event objects solve two problems: first, to check compatibility of nodes during connection (see section 4.2) and, second, to encapsulate the data for communication between nodes. All events are derived from the base class `Event` which provides the attributes and operations shown in the class diagram in figure 3.5.

Event

`stamp: int`
`delay: int`
`weight: double`
`sender: Node*`
`receiver: Node*`

`deliver(): void`

Figure 3.5: UML diagram for class `Event`: The most important member variables (top part) and functions (bottom part) of class `Event`.

The different types of nodes that were introduced above need different events to transmit their information to the target nodes. A node can provide event handlers each available event by overloading the corresponding `handle()` function (see section 3.1.1). However, nodes are restricted to a single event type that they can send. This restriction is necessary to satisfy the requirements of the type checking algorithm that is executed during connection setup (section 4.2).

In their `deliver()` function, the standard event types call `handle()` on the receiving node with a reference to itself as argument. However, some nodes need to send different data for each node, e.g. the Poisson generator transmits a random number of spikes to each of its targets. To support this mechanism, special events (*direct sending events*) are available that do not call `handle()` on the receiver, but `event_hook()` on the sender, with the receiver as argument. This way, the sender is informed about each of its targets for every delivery it has issued. The different routes for the delivery of events are described in chapter 5.

Chapter 4

Connection Management

The connection framework of NEST2 has been improved to support not only static connections, but different types of plasticity and learning in neuronal networks. The main class of this new system is the `ConnectionManager`, which has the following tasks:

- Store the connection data and allow dynamic changes.
- Provide different synapse types.
- Administrate default values for each synapse type.
- Calculate auxiliary variables during connection setup.
- Provide a convenient SLI interface to the user.

The connection data of the `ConnectionManager` is stored in so-called `Connector` objects. Additionally, this is the place, where learning algorithms, which change the connection parameters, are implemented. Each `Connector` is created by a `ConnectorFactory` that is the *prototype* for a synapse type. The prototypes hold default values and provide the SLI interface to retrieve and set them. The UML diagram in figure 4.1 shows the `ConnectionManager` and its components.

The `ConnectionManager` stores a list of synapse *prototypes* that are available to connect nodes to each other. The type for a connection can be selected in SLI by using a prototype's unique *synapse id*, which corresponds to its position in the list. Instead of specifying the type for each connection separately, a default synapse type can be set, which is stored in the variable *synapse context*. Connections are stored by the `ConnectionManager` in a three-dimensional data structure, *connections* (see figure 4.3), in which the innermost array contains `Connector` objects that actually store the connection information. Its dimensions represent

1. The thread number of the target node of the connection.
2. The global id of the source node of the connection.
3. The index of the synapse prototype of the connection.

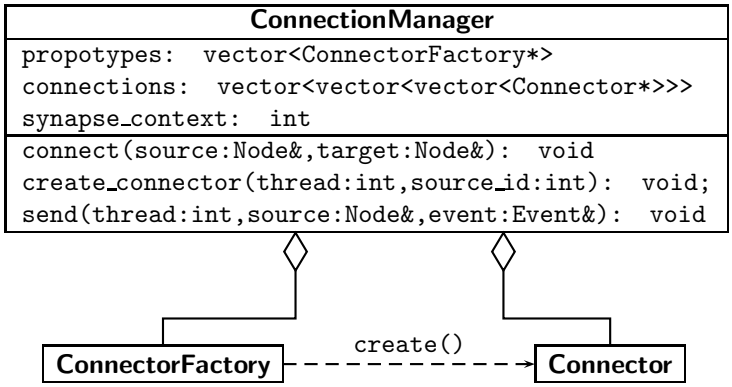


Figure 4.1: UML diagram for class `ConnectionManager`: The most important member variables (top part) and functions (bottom part) of class `ConnectionManager` together with its main components, `Connectors` and `ConnectorFactories`.

4.1 Connectors and Connection Prototypes

Synaptic plasticity changes the connection parameters during simulation. The rules for these changes are implemented inside of `Connector` objects, which also store the connection information. Each connection consists of a *source* and a *target* node together with a connection *weight* and a *delay*. Additionally, a *receptor port* can be set on the connection to select the response of the target, e.g. to provide different time constants or to build compartment neuron models. Different types of `Connectors` are available that derive from a common base class. The interface for this base class is shown in figure 4.2. The variables *min_delay* and *max_delay* are used to store the local extrema determined during connection setup (see section 4.2.3).

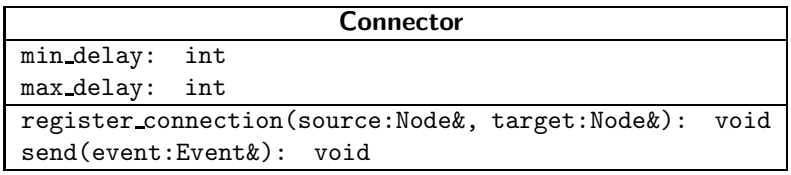


Figure 4.2: UML diagram for class `Connector`: The most important member variables (top part) and functions (bottom part) of class `Connector`.

In order to reduce memory overhead in the new system, `Connectors` are only created for nodes that actually have connections of this type. They are created by the corresponding `ConnectorFactory`, which serves as prototypes a particular type of synapse. The prototypes are created in the initialization process of the simulator and stored in the `ConnectionManager`'s *prototypes* list. In addition to the creation of `Connectors`, they are used for the administration of default values and several auxiliary values for this specific type of synapse. The user can parameterize the synapse types and store them with a new name and id for later use during simulation setup. A mapping of *synapse id*s to descriptive names is provided in the *synapsedict* dictionary in SLI. In section 4.3, we will present two small examples of the SLI interface to the

new connection framework.

The connection system of NEST2 allows the same flexibility as the system in Paranel, but only the static synapse type of NEST has been re-implemented as `Connector` for the new system. It only supports the basic connection parameters *sender*, *receiver*, *weight*, *delay* and *receptor port* and does not implement any dynamic behavior. Currently, work is done to implement a `Connector` for synaptic short term dynamics as described in Tsodyks et al. (1998). Preliminary results are that the system behaves as expected and the findings from Tsodyks et al. (2000) can be reproduced with NEST2.

4.1.1 Data Compression

The simulation of biological neuronal networks requires a lot of memory. Distributed simulation allows to simulate networks larger than one machine's memory by using multiple computers, but does not reduce the overall memory consumption of a simulation. An important property of Paranel's connection system is its ability to compress connection data. In Paranel, several synapse types exist that do this by storing common values only once. This way, redundancy in the network can be exploited to reduce the required memory an thus to enable the simulation of larger networks. With the new connection manager, this is also possible in NEST2. A discussion of different compression schemes can be found in Morrison et al. (2005).

4.2 Establishment of Connections

In this section, we illustrate how two nodes are connected. In the simplest case, a new connection is established by calling the SLI command `Connect`. It takes only two argument: the global identifiers of the source and target nodes. Several high-level routines for the connection of whole populations are based on this simple version and will be described in section 4.3.1. The algorithm `connect()` is shown as pseudocode in listing 4.1.

```
1  void ConnectionManager::connect(Node& source, Node& target)
2  {
3    if (target.type = proxnode)
4      return;
5
6    tid ← target.thread;
7    sid ← source.id;
8    source ← network.get_node(sid, tid);
9    if (connections[tid][sid][sc] = 0)
10     create_connector(tid, sid);
11   connections[tid][sid][sc].register_connection(target);
12 }
```

Listing 4.1: The connection algorithm. New connections are registered with the `Connector` at position (*tid*, *sid*, *sc*) in the *connections* structure. *tid* is the thread of the target; *sid* is the global id of the source; *sc* is the current synapse context.

If the target node of a connection is a proxy, the `connect()` function will return. This behavior is important for distributed connection setup and will be explained in section 4.2.2.

Line 8 calls the function `Network::get_node()` to obtain a pointer to the source node in the thread of the sender. This is necessary, to connect neurons and devices on the same thread. The position of the `Connector` in the *connections* structure is determined by using the *thread* of the target, the *global id* of the source and the current *synapse context* (see figure 4.3). If no `Connector` exists at that position, the corresponding `ConnectorFactory` is used to create it via `create_connector()`. This way, the size of the *connections* structure is kept minimal, because `Connectors` are created dynamically as needed. It has to be noted that `connections` is thread safe since it contains separate `Connectors` for each thread. This is important during simulation, when multiple threads are used (see section 5.8).

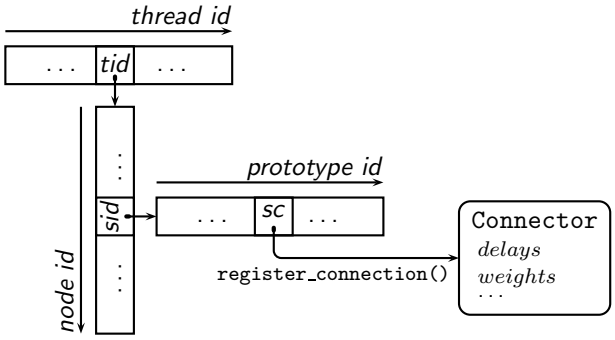


Figure 4.3: The connection structure: New connections are registered with the `Connector` at position (*tid*, *sid*, *sc*). *tid* is the thread of the target; *sid* is the global id of the source; *sc* is the current synapse context.

In the previous chapter we have shown that virtual processes can be represented without using proxy nodes for local nodes; proxy nodes are only required for nodes living in remote processes. This has helped to reduce the memory requirements in multithreaded setups. For the connection system, they are re-introduced and as anchors for connections which would otherwise join neurons on separate threads. In this way, from the perspective of an individual thread, all connections appear to be local, thus minimizing collisions of threads and thus cache problems (see section 2.2.5). Because the connections are stored separate from the nodes, no proxy nodes have to be allocated. It is sufficient to keep the size of the node list in *connections* (see figure 4.3) equal to the size of `Network`'s node list.

4.2.1 Type Checking

To make sure that all connection targets can handle the events they will receive during simulation, `register_connection()` implements a type checking algorithm that is executed during connection establishment. From the kernel's point of view this type check is static, meaning that once established, the event type of a connection cannot change. From the C++ point of view, it is dynamic, because the type check is not carried out during compilation. The compatibility of source and target is ensured by `Connector::register_connection()` by calling `check_connection()` on the source (see listing 4.2), which then calls `connect_source` on the target. The sequence diagram for `check_connection()` is shown in figure **??**. In the base class, `connect_source()` returns *false* but may be overloaded in derived node classes for each

type of event that the node has to support. Because the `update()` function of a node can only send a single type of event (`EventT`), `check_connection()` needs not be overloaded for all available events, but only has to test a single type.

```
bool Node::check_connection(Node& target)
{
  EventT event;
  event.set_sender(this);
  event.set_receiver(target);
  return target.connect_sender(event);
}
```

Listing 4.2: Static connection checking. `EventT` represents an event of arbitrary type that depends on the actual node

If `connect_source()` returns *true*, the target accepts the connection and the connection information is stored in an appropriate way by the `Connector`.

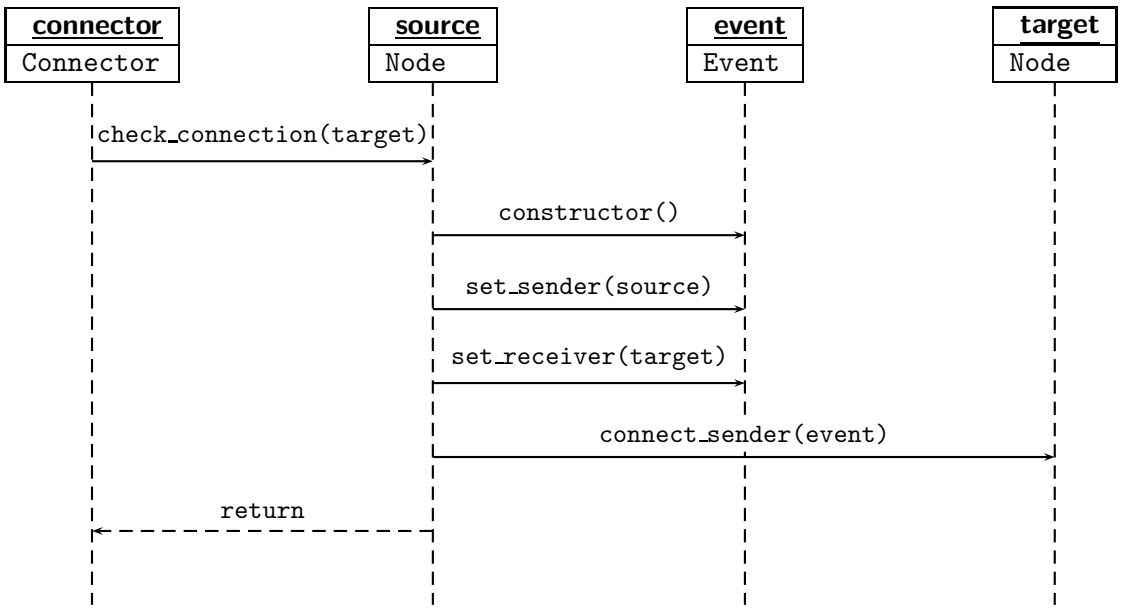


Figure 4.4: Sequence diagram for the connection type check.

4.2.2 Distributed Connection

If we want to simulate a neural system with many nodes and connections, it may take longer to create and connect all neurons that it takes to simulate the network. An obvious solution is to also construct the network in parallel. Each process only registers connections where the target node is not a proxy node. On computers where the target is a proxy node the

connection is ignored. This way the distribution of the nodes leads to a distribution of the connection information in a natural way and additionally reduces connect time. The following listing shows again lines 3 and 4 from the connection algorithm (see listing 4.1) and contains the code to determine whether a connection has to be established in a particular process or not.

```
3 if (target.type = proxnode)
4   return;
```

There are two possible ways to distribute the connection information onto multiple processes in a distributed simulation setup: Either the connections are stored in the process of the presynaptic node or in the process of the postsynaptic node. The distribution directly influences the amount of data that has to be sent during the simulation. If we assume a network where the number of the connections k is a certain fraction c of the number of nodes n, the following estimation of the number of events holds:

1. Presynaptic storage: For each of the nodes we have to store a target list that contains all its connections. Let λ be the mean firing rate of the node and k the mean number of connections a node has, $k = cn$. Then each process has to transmit $\lambda \cdot n \cdot k$ events to other machines. For k approaching n this results in $O(n^2)$ events. This is the case for $n < 10^5$ in biological networks. For $n > 10^5$, the number of connections scales linearly with the number of nodes and results in $O(n)$ events.
2. Postsynaptic storage: Instead of sending all events to remote processes, the node only sends the information about the event once. The postsynaptic process is then able to reconstruct the event locally and deliver it to the local targets. This means that for m the number of machines, only $\lambda \cdot n \cdot m$ events are necessary. Assuming that m is usually small compared to k, this results in an overall number of events of $O(n)$.

The lower communication volume was the reason to prefer the second solution for the implementation of the new connection system of NEST2. Moreover, some kinds of synaptic plasticity, like e.g. spike-timing dependent plasticity (e.g. Morrison et al., 2005) depend heavily on the dynamics of the postsynaptic node, which also militates for this scheme. However, things like presynaptic homeostasis (e.g. Fregnac, 1998) where the presynaptic neuron normalizes its output are rather difficult to implement with this form of storage.

In figure 4.5 a small network is shown to illustrate where the connections are stored in a setup with two processes with two threads each.

4.2.3 Calculation of the Minimal and Maximal Connection Delay

For the network calibration (see section 5.4), performed before the simulation, two parameters are of importance: $d_{\min}$, the minimal connection delay in the network and $d_{\max}$ accordingly. These are used to calculate the sizes of the ring buffers (see section 5.4) for event buffering. As the connections are registered with the `Connectors`, these have to calculate the new values for $d_{\min}$ and $d_{\max}$ by comparing them to the old values. Right before the simulation is run, the `ConnectionManager` collects the local extrema and calculates the global values that are used during the simulation. In a distributed scenario, the outcome of each process' calculation may be different, because each of them only knows about the local connections (section 4.2).

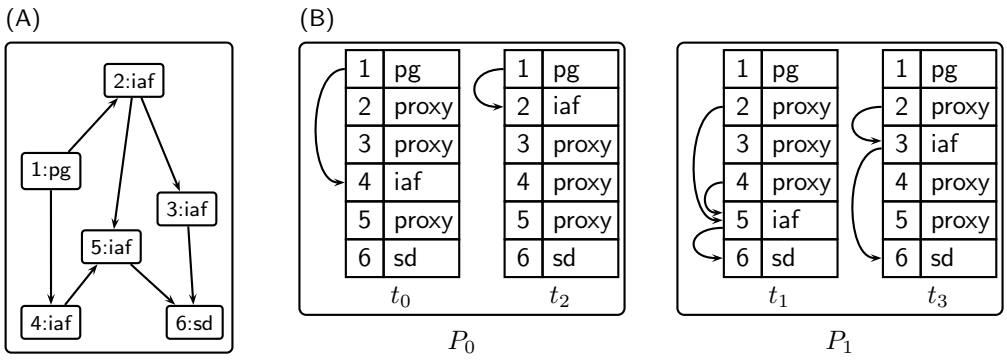


Figure 4.5: Connections in a distributed simulation: A small example of a neural network consisting of 6 nodes: 1 Poisson spike generator (pg), 4 integrate-and-fire neurons (iaf), and 1 spike detector (sd). (A) The network as a directed graph. The edges of the graph contain a node's *global id* and its *type*, separated by a colon. (B) The connectivity of the network distributed on two processes with two threads each. Connections are only established on threads, where the target node is not a proxy.

To solve this problem, each process has to send a message containing its values to all other processes to determine the global values.

4.3 The SLI Interface

The SLI interface to the new connection framework will be introduced by presenting two simulation scripts that make use of the new commands and data structures. First, an overview of the variants of the new `Connect` commands is given.

4.3.1 Connection Functions

SLI provides several functions to connect nodes to each other. All of them are built on top of a single connection function in the kernel (see section 4.2).

`Connect` takes the global identifiers of two single nodes and connects them (figure 4.6 (A)).

`ConvergentConnect` takes an array of source nodes and a single node. A connection is established from each of the source nodes to the target node (figure 4.6 (B)).

`DivergentConnect` takes a single node and an array of target nodes. A connection is established from the source node to each of the target nodes (figure 4.6 (C)).

4.3.2 An Example SLI Session

In this section, we will provide a small overview over the commands and data structures of the new connection framework. Instead by running a script, the commands are executed interactively at the SLI command line prompt, which is indicated by '`SLI ]`'.

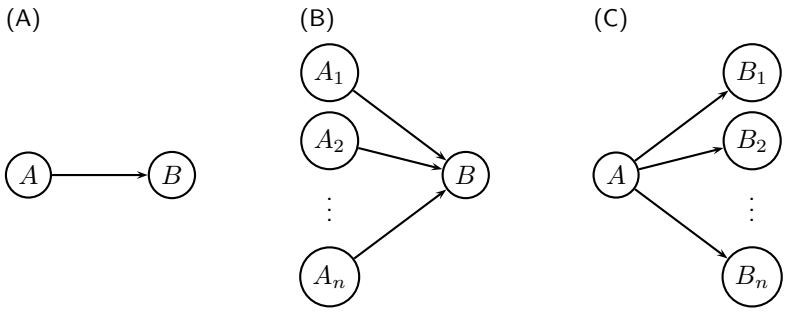


Figure 4.6: The different connection functions: (A) Standard connect: connect two nodes. (B) Convergent connect: connect a population to a single node. (C) Divergent connect: connect a single node to a population of nodes. The routines for convergent and divergent are also available in a randomized version that expects the number of connections to be drawn from the addresses of the population as additional argument.

First, we use the command `info` to inspect the contents of the dictionary for the synapse types.

```
SLI ] synapsedict info
```

Name	Type	Value
static_synapse	integertype	0

Total number of entries: 1

Now, we print out the current *synapse context* to verify it is set to the only available synapse type.

```
SLI ] GetSynapseContext
0
```

In the next step, we want to see the default values for the synapse prototype. Note that the synapse type is given implicitly by the synapse context. `GetSynapseDefaults` returns a dictionary, which we can print again using the `info` command.

```
SLI ] GetSynapseDefaults info
```

Name	Type	Value
weight	doubletype	1
delay	doubletype	1.0
receptor_port	integertype	0

Total number of entries: 3

The following sequence of commands creates 2 integrate-and-fire neurons and connects them to each other using the currently active synapse prototype. We verify the connection by printing the target list of neuron 1.

```
SLI ] iaf_neuron 2 CreateMany pop
SLI ] 1 2 Connect
SLI ] 1 GetStatus /connections get /targets get ==
[2]
```

By using `SetSynapseDefaults` we can set new default values, here, the default weight is set to 0.5. Again, the command implicitly applies to the current synapse context. The new connection is automatically assigned the new weight, which we confirm by printing the *weights* list for neuron 1.

```
SLI ] << /weight 0.5 >> SetSynapseDefaults
SLI ] 1 2 Connect
SLI ] 1 GetStatus /connections get /weights get ==
[1.0 0.5]
```

4.3.3 Building and Connecting a Small Network

To show the functionality in a more realistic setup, we build the example network that has been used throughout the description of the different network representations in chapter 2. For convenience, the network is shown again in figure 4.7

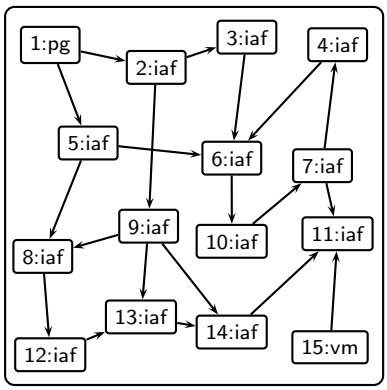


Figure 4.7: A small example of a neural network: The network consists of 15 nodes: 1 Poisson spike generator (pg), 13 integrate-and-fire neurons (iaf), and 1 voltmeter (vm). The edges of the graph contain a node's *global id* and its *type*, separated by a colon.

We start by creating all nodes in a row. The nodes are assigned assigned consecutive *global id*s, starting with 1 for the first element, a Poisson spike generator. The `Create` command returns a handle to the new node, which we store in variables for later use. `CreateMany` returns a handle only to the last node that it created, which we discard by using `pop`.

```
poisson_generator Create /pg Set
iaf_neuron 13 CreateMany pop
voltmeter Create /vm Set
```

The structure of the following code block basically resembles the adjacency list of the network, which was shown in figure 2.2. Nodes that have multiple targets are connected using `DivergentConnect` for convenience.

```
pg [2 5] DivergentConnect
2 [3 9] DivergentConnect
3 6 Connect
4 6 Connect
5 [6 8] DivergentConnect
6 10 Connect
7 [4 11] DivergentConnect
8 12 Connect
9 [8 13 14] DivergentConnect
10 7 Connect
12 13 Connect
13 14 Connect
14 11 Connect
vm 11 Connect
```

Chapter 5

Simulation

5.1 Strategic Considerations

To control the flow of time in computer simulations, two different paradigms are are known: *time-driven* and *event-driven* (Zeigler et al., 2000). Both are illustrated in figure 5.1. In the first approach, each element is updated on a time grid with fixed spacing, thus time is driving the simulation. This is a common strategy for simulations of physical systems. All elements are updated in every time step, regardless if their state has to be changed or not. This may pose an unnecessarily great load on the simulation machinery if a large number of elements is simulated. The second approach is inspired by the fact that the elements cannot be influenced by one another unless an interaction takes place between them. It is sufficient to update the elements only after an event has arrived and needs to be processed. The flow of time in this scheme is defined by the order of the events. It is obvious, that the event-driven approach is most advantageous if the number of events is small in comparison to the overall number of elements. This is, however, not the case with neural simulations, where the number of connections that have to carry the events outnumbers the number of elements by several orders of magnitude.

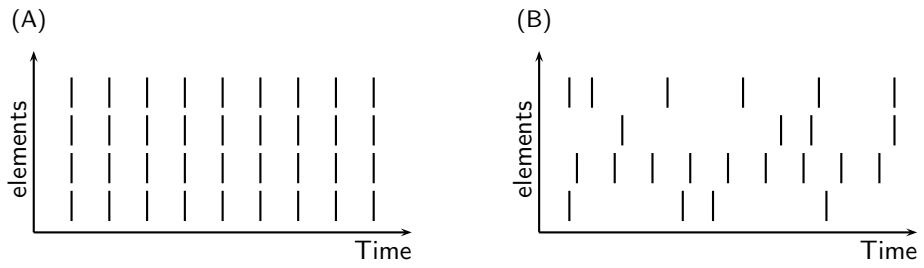


Figure 5.1: Schematic comparison of simulation strategies: Modified from Fujimoto (2000). A vertical bar indicates a state change of an element. (A) Time driven: each node is updated on each grid point. (B) Event driven: the nodes are only updated if an event occurs; Note that state changes are not forced to a fixed time grid.

NEST implements a mixture of time- and event-driven simulation. The neural network is

simulated on a time grid with fixed spacing. The the elements are updated in a loop which is shown in the flow chart in figure 5.2. In the distributed case, the cycle runs independently in all processes and is synchronized after each update cycle. For an elaborate discussion of update strategies, see Morrison et al. (2005).

The state of the network is updated by calling `update()` on each node that is not in the *frozen* state. In each update cycle, the simulation time is advanced by $d_{\min}$ time steps. The events that are generated during the cycle are buffered for later delivery. To make the local event buffers available on all remote machines in a distributed simulation setup, the event buffers are exchanged between the processes after all nodes have been updated. Finally, the network time is advanced and the events that were buffered beforehand are delivered to their local targets on each process. The loop is repeated until the specified simulation time has elapsed.

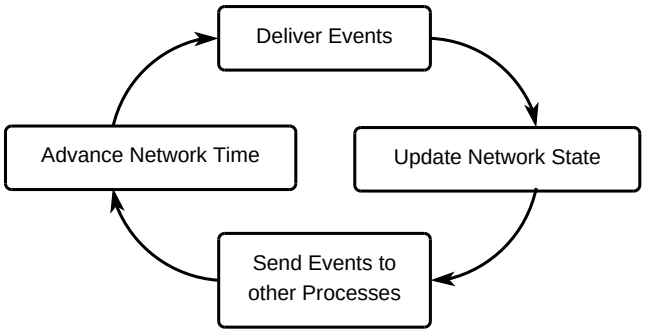


Figure 5.2: The NEST2 simulation loop.

In the following discussion, the serial case will be omitted for clarity. One can always think of it as multithreaded with one thread. The implementation, however, provides different algorithms for these cases to avoid possible overhead.

5.2 The Scheduler of NEST2

The algorithms for the control of the simulation and for communication are implemented in class `Scheduler`. Its class interface is shown in figure 5.3.

5.3 Definitions

The smallest possible interval of time that can occur in a network is denoted by h and referred to as a *time step*. The minimal synaptic propagation delay in a network is called $d_{\min}$. It is expressed as integer multiple of h and called a *time slice*. Let n be the number of the time slice. Then time slice n begins at $T_n^0 = n \cdot d_{\min}$ and ends at $T_n^\infty = (n+1) \cdot d_{\min} - 1$. In analogy to $d_{\min}$, $d_{\max}$ is defined as the largest connection delay in the network. Both values are determined by the `ConnectionManager` (see section 4.2.3). The above definitions are depicted in the time diagram in figure 5.4.

Scheduler
spike_register: vector<vecor<vector<int>>> comm_buffer: vector<vector<int>> partners: vector<int> nodes: vector<vector<Node*>>
calibrate(): void send_remote(thread:int, event:Event&, lag:int): void communicate() : void

Figure 5.3: UML diagram for class `Scheduler`: The most important member variables (top part) and functions (bottom part) of class `Scheduler`.

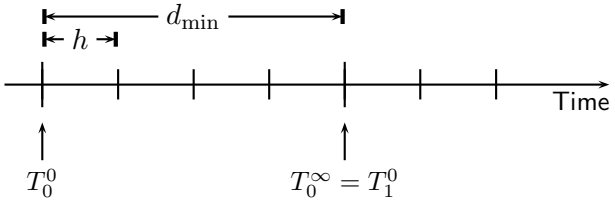


Figure 5.4: Definitions of time in NEST: $d_{\min}$ is the minimal connection delay in the network, expressed in integer multiples of h. The nth time slice of length $d_{\min}$ begins at T_n^0 and ends at T_n^∞.

5.4 Network Calibration

The scheduler must be re-calibrated every time the simulation grid size (h), the number of threads (t), the number of nodes (n) or the global extrema of for the connection delays (i.e. $d_{\min}$ and $d_{\max}$) change. The `calibrate()` function executes the following steps:

1. It broadcasts the local $d_{\min}$ and $d_{\max}$ to all processes and determines their global values.
2. It computes frequently used constants, so they don't need to be re-computed during the simulation
3. It calls the `calibrate()` function of each node to adjust the node's parameters and to compute frequently used constants. To avoid cache related problems during simulation, the calls to this function are organized in blocks of nodes that belong to the same thread.
4. For each thread t it fills the buffer *nodes*$[t]$ with references to the nodes the thread must update.

Calibration of Nodes

Ring buffers (see section 3.1.2) store events that have been received by a node, but that are not yet due. The size of the ring buffers depends on $d_{\min}$ and $d_{\max}$. Thus the memory for the ring buffers can only be allocated after all nodes are connected.

The capacity of the ring buffers is determined by the earliest and latest delivery time of an

event together with the minimum and maximum delay as follows:

- Let 0 be the current ring buffer position, i.e. the position we read from. Events sent by a device are sent directly and the earliest arrival time will be 1, the latest $d_{\max}$.
- Events that are sent by neurons are buffered in the *spike_register* and will be delivered earliest at $d_{\min}$, latest at $d_{\min} + d_{\max} - 1$.

Combining both ways of event transmission, we need a capacity of $d_{\min} + d_{\max}$ buffer elements. The different treatment of events originating from devices and neurons is explained in the following section.

5.5 Network Update

For every given point in time, the elements in a network are functionally decoupled for a time interval of $d_{\min}$. This is used by Paranel in two places: in the node update and in the communication between processes. For the NEST2 scheduler, the update algorithm of Paranel has been adapted to exploit the processor cache optimally and limit the number of memory accesses. It is generally best to compute as much as possible with the data already available in the processor's cache. This can be achieved by updating every node $d_{\min}$ times, before moving on to update the next node. Internally, the node may well use a complete different time scale to propagate its internal state. This is one of the main differences between NEST and the NEST2 kernel, where the network has been updated only by one time step h in each cycle. This means that previously h played the role of the simulation resolution and defined the grid for network update. In NEST2 it is only used as the default computation time step onto which the occurrence of events is forces. However, in the work of Morrison et al. (2005) a scheme for precise spike timing in grid-based simulations of neuronal systems is presented where an additional floating point number in the event is used to represent fractions of h and thus achieve sub-grid resolutions.

The update algorithm of NEST2 is implemented in the `Scheduler`'s `update()` function (see listing 5.1), which is called once in every update cycle by each thread with its id t as argument. and advances all nodes in the thread's buffer by $d_{\min}$ time steps.

```
void Scheduler::update(int thread)
{
  for (int nid = 0; nid < nodes[thread].size(); nid++)
    nodes[thread][nid].update(slice_begin, slice_begin + d_min);
}
```

Listing 5.1: The update algorithm of NEST2. This function has to be called once in every update cycle by each thread. It updates the nodes of the thread by every node $d_{\min}$ in each call.

NEST dynamically assigns nodes to threads in each cycle. This leads to a high probability that a node is updated by different threads in each successive update cycle. The consequence of this is cache thrashing, which results in bad performance because the content of the processor's caches is replaced in each cycles. NEST2 does not have this problem, because each node is

assigned to exactly one thread for the entire simulation. The strict assignment of nodes to threads has the additional advantage, that the ring buffers do not have to be thread safe, which also reduces the risk of cache thrashing.

Dispatching Events

If a node generates an event in its `update()` function, it uses `Network::send()` to deliver the event to its targets (see listing 3.1). Events from neurons and devices are treated differently, because devices only can have local targets, whereas the events of neurons also have to be sent to the remote processes (see section 4.2.2). The route an event take is chosen in `Network::send()`, which is shown in listing 5.2.

```
void Network::send(Node& source, int thread, EventT& e, int lag)
{
  e.set_stamp(slice_begin + lag + 1);
  e.set_sender(source);

  if (source.type = device)
    connection_manager.send(thread, e, source);
  else
    scheduler.send_remote(thread, e, lag);
}
```

Listing 5.2: The function for sending events in NEST2. At the beginning, the time and sender are set on the event. Devices send their events directly via the `ConnectionManager`, events by neurons are buffered by the `send_remote` function of the `Scheduler`.

Direct Sending of Events

The function `send()` of class `ConnectionManager` will, based on a given thread number and the global identifier of the source node, forward the event to each `Connector` of the node. The `Connector`'s `send()` function is shown in listing 5.3. It works by first setting the parameters *receiver*, *weight*, and *delay* and delivers the event by calling its `deliver()` function. As explained in section 3.2, the default implementation of `deliver()` will call handle on the target, which then processes the event. The sequence diagram in figure 5.10 shows the steps that are necessary to deliver an event from a sender to all its targets.

```
void Connector::send(Event& e)
{
  for (int i = 0; i < targets.size(); i++)
  {
    e.set_receiver(targets[i]);
    e.set_weight(weights[i]);
    e.set_delay(delays[i]);
    e.deliver();
  }
```

```
}
```

Listing 5.3: The function for sending events in NEST2. At the beginning the connection parameters are retrieved from the `Connector`'s representation and set on the event. Then the event is delivered by calling its `deliver()` function.

The delivery algorithm explained above sends the same event to each target. Some devices, however, have to send unique events for each of their targets. An example is the Poisson generator, which has to draw a random number of spikes for each of its targets. Because the nodes do not know about their targets in the NEST2 kernel, this is not easily possible. To solve this problem, a special type of event has been implemented. So-called *direct sending events* call `event_hook()` on the sender in their `deliver()` function. This allows the receiver to change parameters of the event and re-send it by calling `handle()` on the target node itself. This is illustrated in the following sequence diagram (figure 5.5).

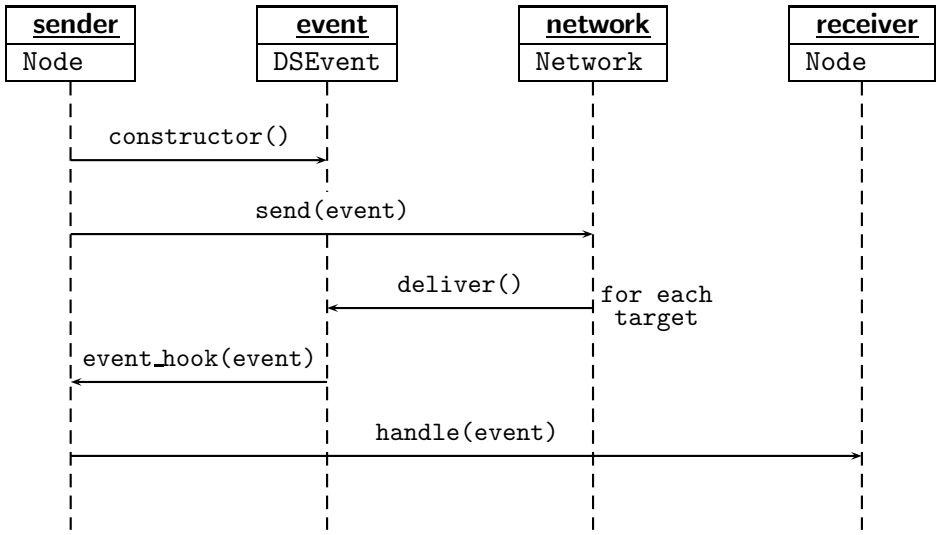


Figure 5.5: Sending a *direct sending event* via the `event_hook()`.

Event Buffering

In contrast to the events that originate from devices, the events from neurons have to be sent to local *and* remote targets and are buffered in the *spike_register*. This is a three-dimensional data structure where the dimensions represent:

1. The lag inside the time slice.
2. The thread number of the sending node.
3. The event number.

Events are stored in the *spike_register* by the `send_remote()` function of the `Scheduler` (see listing 5.4). An example call to `send_remote()` is illustrated in figure 5.6.

```
void send_remote(int thread, Event& event, int lag)
{
  spike_register[lag][thread].push_back(event.sender.id);
}
```

Listing 5.4: Buffering events for local and remote delivery. At the beginning the event parameters are set. Then the event is delivered by calling its `deliver()` function.

Note that only the *global id*s of the sending nodes are stored in the *spike_register*. This restricts the current implementation to the use of `SpikeEvents` for neuron-neuron communication. However, this limitation reduces the communication volume and improves the performance of the program. Moreover a trivial solution exists, namely to buffer and send the complete events. This is also discussed in chapter 7.

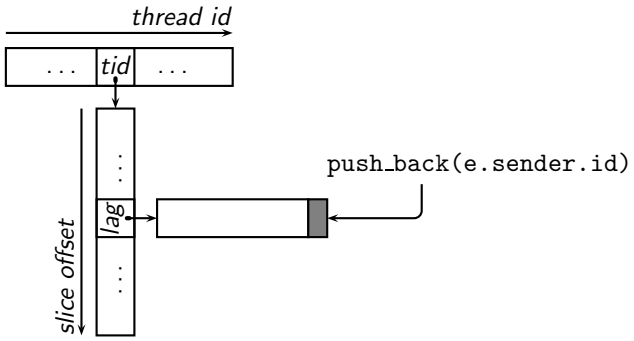


Figure 5.6: Illustration of event buffering: A call to `Scheduler::send_remote(lag, tid, e)` appends the global id of the sender to the corresponding list of the *spike_register*.

5.6 Communication

After the network has been updated, each process has to send its spike information to all other processes for local delivery. Because the number of processes in a simulation is constant, the communication partners can be determined before the simulation. Each process is characterized by a unique number, its *rank*. The number of the partners are determined by the algorithm for Complete Pairwise EXchange (CPEX, Tam & Wang, 2000) and are stored in the *partners* array of class `Scheduler`. Each process executes the CPEX algorithm independently and determines its own partners. The partner for step i is stored at position i in *partners*. If the number of processes is odd, one of the processes remains idle in each step and -1 is stored in the corresponding field of *partners*. The CPEX algorithm is illustrated for two different setups, once with five, once with six processes, in figure 5.7.

The events that were generated during the update of the network are stored in the *spike_register* as it was shown in figure 5.6. To minimize communication overhead, as few packets as possible have to be sent over the network connection. The number of packets is reduced by collocating the fields of *spike_register* into a single *comm_buffer* that is then sent

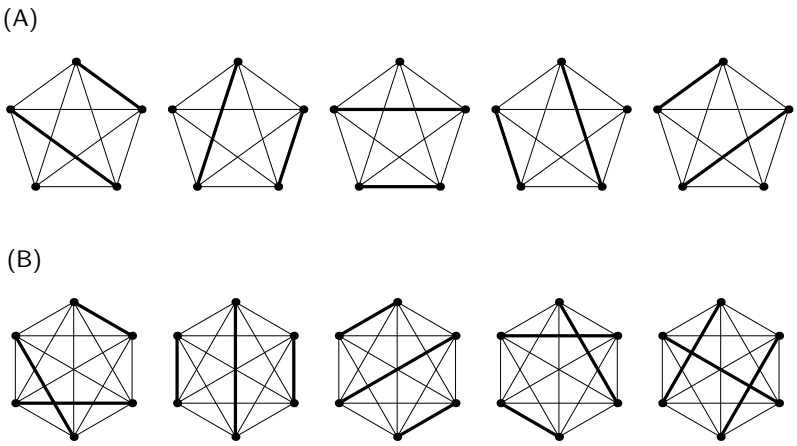


Figure 5.7: Illustration of the CPEX algorithm: Modified from Morrison et al. (2005). Setup with (A) odd and (B) even number of communication partners. Bold lines connect the communcation partners in each step.

to all remote processes by the communication algorithm. To decrease the number of communication steps, the processes exchange their buffers only after an interval of $d_{\min}$, the minimal connection delay in the network. In the *comm_buffer*, the data for the steps of the time slice is separated by markers. The markers are used on the side of the receiver to reconstruct the envents (see section 5.8). The collocation of the *spike_register* into the *comm_buffer* is illustrated in figure 5.8. It is important to carry out this step in the same way for multithreaded and distributed simulation to ensure the same order of event delivery, which then guarantees reproducible results.

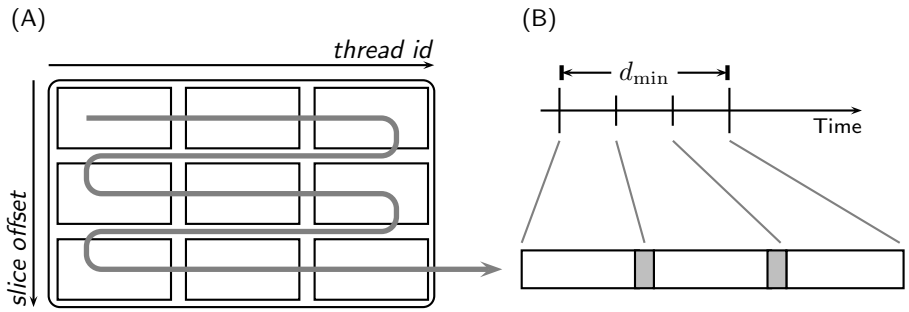


Figure 5.8: Preparation of *comm_buffer*s for inter-process communication: (A) The *spike_register*. The bold gray arrow indicates the direction in which the buffer is collocated. A marker is inserted between each step of the time slice. (B) The *comm_buffer* with markers (gray blocks). Note that each data block represents a list of spikes from all threads.

The *partners* list that has been populated by the CPEX algorithm is used in the function `communicate()` (see listing 5.5) to ensure that only the minimal number of communication steps is executed and that no dead-locks will occur. The communication algorithm consists of

a loop that executes the following steps:

- Assign the current communication partner to the variable *partner*
- If *partner* is valid (i.e. *partner* ≥ 0) the process with smaller *rank* sends its *comm_buffer* to its *partner*, while the other one receives. Then the process with larger *rank* sends and the *partner* receives.

```
void communicate()
{
  for (int step = 0; step < partners.size(); step++)
  {
    partner = partners[step];

    if (partner >= 0)
    {
      if (rank < partner) // smaller process number sends first
      {
        Send(comm_buffer[rank], partner);
        Receive(comm_buffer[partner], partner);
      }
      else
      {
        Receive(comm_buffer[partner], partner);
        Send(comm_buffer[rank], partner);
      }
    }
  }
}
```

Listing 5.5: The communication algorithm of NEST2.

The low level communication between the processes is carried out by an external library that implements the Message Passing Interface (*MPI*, Message Passing Interface Forum, 1994). MPI is a library specification for message-passing that is widely used in high-performance computing applications. It offers an abstracted and portable layer for inter-process communication and several implementations are available for different platforms. In listing 5.5 the use of MPI is indicated by the `Send` and `Receive` functions.

5.7 Time Evolution

After the network was updated and the local event buffers have been exchanged by the processes, the system time has to be advanced. Because each node has been propagated by $d_{\min}$ time steps, the state of the clock is increased by the same amount.

5.8 Event delivery

The main function for sending events is `Network::send()`. It decides whether the event will be sent only locally or also to remote targets according to the type of the sending node (see listing 5.2).

Devices are created once for every thread in the system, thus they are guaranteed to have only targets local to their thread. Their events may be sent directly to all their targets without the need for threadsafe data structures. This has already been explained in section 5.5 and leaves only the events from neurons to be delivered.

The events of neurons also have to be sent to the remote machines for processing. The `Scheduler` queues all events sent by these nodes. At the end of the update cycle it transmits the events by a call to `communicate()` to remote machines (see section 5.6) so that all *comm_buffer*s are available in each process for local delivery of the events.

In listing 5.6 contains the algorithm for the reconstruction of event time stamps and event delivery. To enhance readability, it is only shown for a single *comm_buffer*. The sequence of the complete readout procedure is depicted in figure 5.9. `deliver_events()` is called by each thread and will only deliver events to the targets that belong to it. The *comm_buffer*s contain only the id of the sending node. This, however is not a problem because the actual connection parameters are stored on the process of the postsynaptic node and the event can thus be reconstructed. The markers are used to calculate the time stamp of the event.

```
void deliver_events(int thread)
{
  lag ← d_min − 1;
  for (int i = 0; i < comm_buffer.size(); i++)
  {
    if (comm_buffer[i] ≠ marker)
    {
      source ← network.get_node(node, thread);
      SpikeEvent e;
      e.set_stamp(clock − lag);
      e.set_sender(source);
      connection_manager.send(thread, e, source);
    }
    else
      lag−−;
  }
}
```

Listing 5.6: The event delivery algorithm of NEST2.

The sequence diagram in figure 5.10 shows the route of an event while it is sent to its target nodes. The `ConnectionManager` and the `Connectors` are omitted to reduce clutter.

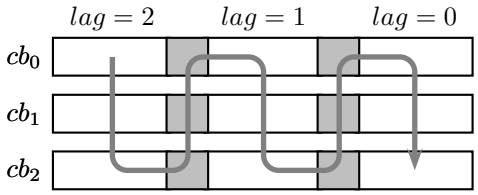


Figure 5.9: Sequence of *comm_buffer* readout: The fields of the *comm_buffer*s are read in sequence of decreasing *lag*. This is indicated by the bold gray arrow.

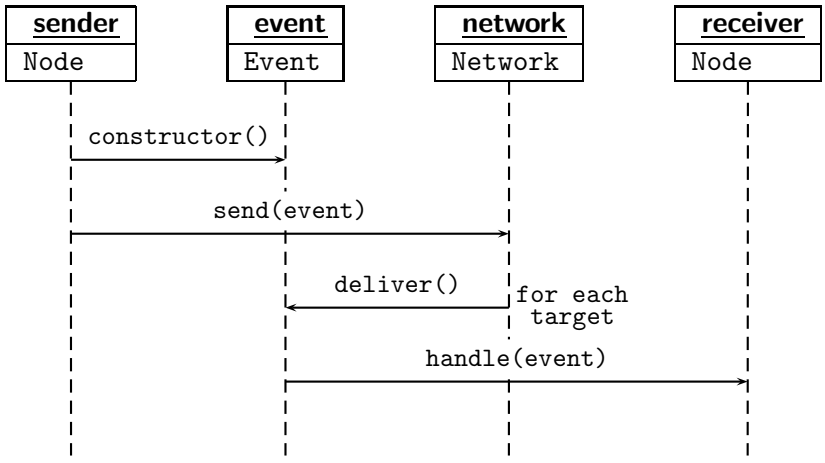


Figure 5.10: Sending an event to its target nodes.

Chapter 6

Performance

A major goal for this work was to improve the overall performance of the kernel by implementing solutions for the problems that have been identified in an analysis of the existing implementations. The result of this is that the new kernel scales nearly as good as Paranel on SMP computers and computer clusters. The following list shows a summary of the specifications of the computers on which the benchmark simulation was run.

- Sun Fire V40z: 4 processors, Dual Core AMD Opteron 875, 2.2 GHz, 1 MB cache
- PC Cluster: 20 $\times$ 2 processors, AMD Opteron 250, 2.4 GHz, 1 MB cache, Dolphin/Scali interconnect

The neuronal network that was used for the benchmarks has 12,500 neurons with 1000 random connections each and an average spike rate of around 2.5 Hz. The network and its dynamics are described by Brunel (2000). For a comparative benchmarks between NEST, NEST2, and Paranel, the same network was implemented once as a C++ program and once as script for the simulation language interpreter.

An important measure for the performance of parallel applications is their *speedup*. Let n be the number of processors. Then the speedup is defined as

$$\begin{aligned} S(n) &= \frac{\text{run time of the serial program}}{\text{run time of the parallel program using } n \text{ processes}} \\ &= \frac{T_1}{T_n} \end{aligned} \tag{6.1}$$

The run times and number of processors in the following sections cover a large ranges of values. It is thus helpful to use logarithmic scales on both axes (*log-log* representation) in the figures. Additionally, this means, that linear scaling will yield a straight line.

6.1 Performance on Multiprocessor Computers

On the SMP machine, the network was simulated for 1 biological second. The absolute run time and the speedup are shown in figure 6.1 for NEST (solid line), NEST2 and Paranel (dashed line). NEST2 supports both, multithreaded (dotted line) and distributed (dash-dotted line) simulations. To compare the two modes, they were used pure, meaning that in the first case a

single process was run with different numbers of threads, in the latter case different numbers of processes each with a single thread.

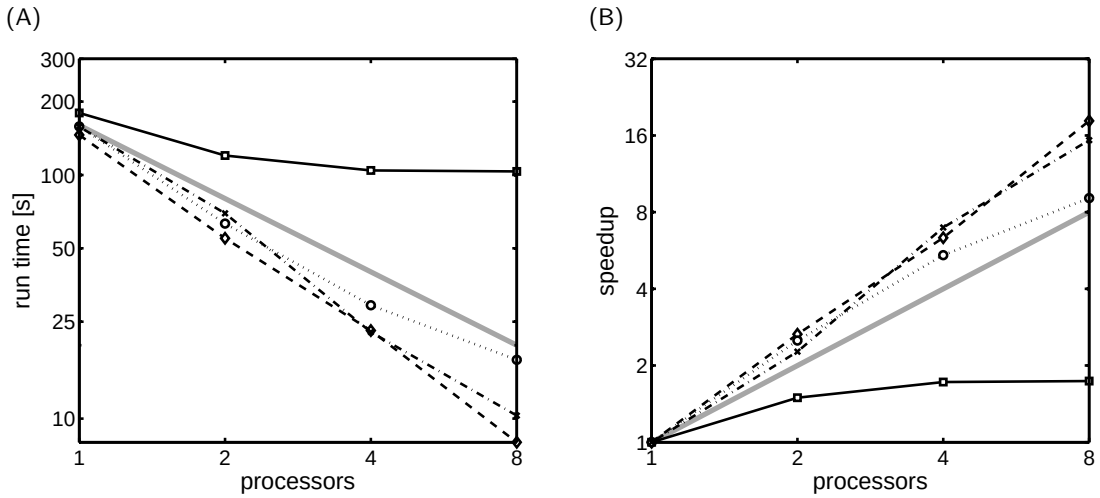


Figure 6.1: Scalability of NEST, NEST2 and Paranel on SMP machines: The network contained 10^4 neurons, with 1000 random connections each; the exact model is described by Brunel (2000). Solid line: NEST; dotted line: NEST2 with multithreading; NEST2 with message-passing; dashed line: Paranel. The gray line indicates linear speedup. (A) Simulation time against number of processors, log-log representation. (B) Corresponding speedup (see equation 6.1) against number of processors, log-log representation.

The NEST kernel shows poor scaling behavior. With more than 2 processors its speedup does not increase further. Moreover, the run time with 2 processors is not even what would be expected from the linear estimation, meaning that with two processors it only is about 1.5 times faster than with a single processor. The scaling can mostly be explained by the memory layout, which does not separate the memory for the threads.

By contrast, Paranel reaches a speedup, that is larger than the linear expectation. This is called *superlinear* scaling and will be discussed in section 6.3. An explanation is that the separation of memory for the processes leads to fewer problems with cache thrashing.

Both simulation modes of NEST2 (multithreading and message-passing) show clear superlinear scaling, which is a great advance over NEST. Although the simulations run equally fast on a single processor, the scaling is still not as good as with Paranel. There are two main reasons for this:

1. Higher memory consumption: device and neurons in NEST2 are larger than the respective elements in Paranel. This leads to more traffic on the memory bus. Thus collisions are more likely.
2. Separation of memory: The memory for the different threads in NEST2 is better separated than in NEST, however Paranel does use different processes that use different memory ranges by definition.

6.2 Scaling on Computer Clusters

Because the run time of the network simulation was already in the range of single seconds with 8 processors on the multiprocessor machine, the simulation time was increased to 50 biological seconds on the PC cluster to get results that are still significantly different when using 40 processors. The results of this simulation is shown in figure 6.2 for NEST2 with message passing (solid line) and Paranel (dashed line).

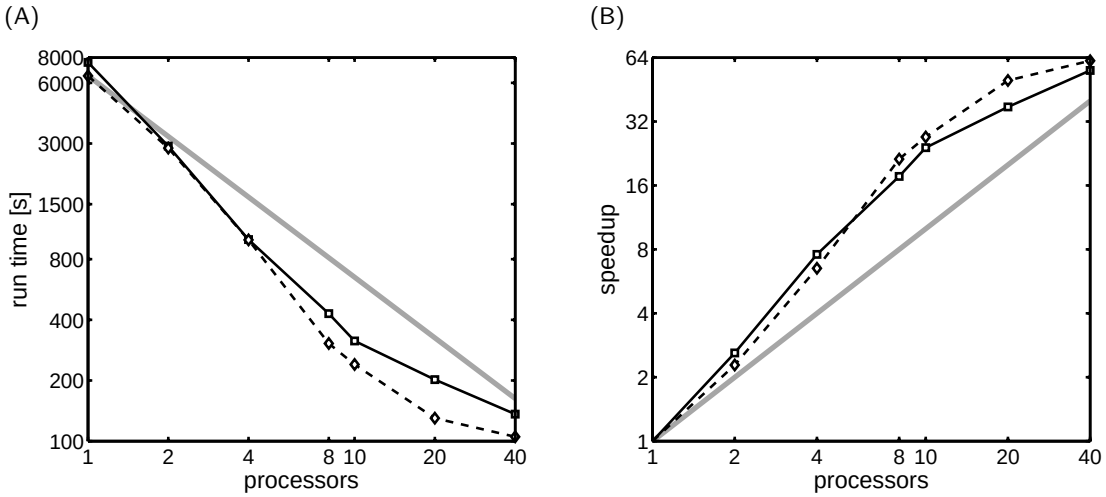


Figure 6.2: Scalability of NEST2 and Paranel on computer clusters: The network contained 10^4 neurons, with 1000 random connections each; the exact model is described by Brunel (2000). Solid line: NEST2 with message-passing; dashed line: Paranel. The gray line indicates linear speedup. (A) Simulation time against number of processors, log-log representation. (B) Corresponding speedup (see equation 6.1) against number of processors, log-log representation.

Again NEST2 and Paranel show clear superlinear scaling. However their speedup returns to the linear above 20 processors. There the complete program is already in the cache and additional processors cannot decrease the simulation time any further. For larger networks, Paranel saturates at a larger number of processor (Morrison et al., 2005). No such simulations were performed with NEST2.

6.3 Cache Effects and Superlinear Scaling

This section contains an analysis of the scalability of parallel programs (modified from Morrison & Diesmann (2003)). The discussion follows the presentation of Wilkinson & Allen (2004). Introductory material can also be found in Tanenbaum (1999). In equation 6.1, the speedup has been defined as

$$S(n) = \frac{T_1}{T_n},$$

where T_1 is the execution time of the serial algorithm of the program and T_n the execution time of the parallel implementation of the program, executed on n processors. Thus, $S(n)$ is the relative increase in speed that is achieved by using n processors. This measure combines characteristics of the software and properties of the computer architecture the program is executed on. In most cases, no serial version of the program is available and T_1 is given by the execution time of the parallel program on a single processor. The speedup is used to evaluate the quality of a parallel program and to investigate the theoretical and practical limits of parallelization. It appears that the maximum speedup is achieved when the problem can be divided into n equal parts without any overhead:

$$\begin{aligned} S_{\text{lin}}(n) &= \frac{T_1}{T_1/n} \\ &= n. \end{aligned}$$

This behavior is called *linear* speedup. In practical applications we expect *sublinear* speedup, because only a part of the program can be parallelized.

Let T_p denote the part of the total execution time that can be parallelized and T_s the remaining sequential part. Then the run time of the serial program is given by

$$T_1 = T_s + T_p$$

Constraints of the operating system like the startup time of a process may also contribute to T_s. In addition, a parallel program typically introduces overhead, like e.g. the communication between processes, which does not appear in the serial version. If we neglect the overhead introduced by parallelization, we can write

$$\begin{aligned} S_{\max}(n) &= \frac{T_s + T_p}{T_s + T_p/n} \\ &= \frac{1}{\frac{T_s}{T_s+T_p} + \frac{T_p}{T_s+T_p} \cdot \frac{1}{n}}. \end{aligned}$$

Substituting the serial fraction of the run time with f,

$$f = \frac{T_s}{T_s + T_p},$$

we obtain Amdahl's law:

$$S_{\max}(n) = \frac{1}{f + (1-f) \cdot \frac{1}{n}}.$$

Thus, there is an upper limit for the speedup that is determined by the serial fraction of the program,

$$\lim_{n\to\infty} S_{\max} = \frac{1}{f},$$

independent of the number of processors. Consequently, with e.g. a fraction of 5 % serial code the speedup is limited to 20. In this model, linear speedup ($S_{\max}(n) = n$) can only be achieved with $f = 0$. Wilkinson & Allen (2004, page 27) comment on this as follows:

> "If the parallel algorithm did achieve better than n times the speedup over the current sequential algorithm, the parallel algorithm can certainly be emulated on a single processor, which suggests that the original sequential algorithm was not optimal."

However, the authors point out that under certain conditions *superlinear* speedup may occur:

> "*Superlinear speedup*, where $S(n) > n$ may be seen on occasion, but usually this is due to using a suboptimal sequential algorithm or some unique feature of the architecture that favors the parallel formation. One common reason for superlinear speedup is the extra memory in the multiprocessor system. For example, suppose the main memory associated with each processor in the multiprocessor system is the same as that associated with the processor in a single processor system. Since the total main memory in the multiprocessor system is larger than that in the single processor system, and can hold more of the problem data at any instant, it leads to less, relatively slow disk memory traffic."

The same consideration should hold if the main memory is compared to the cache memory (Tanenbaum, 1999). Assuming that the data fits into the main memory of a single processor, no advantage is gained by increasing the total amount of main memory. However, with increasing number of processors the total amount of cache memory also increases. The cache memory is distinguished from main memory by faster access times. Therefore, for an arbitrary piece of data the expected access time should decrease with an increasing number of processors.

Chapter 7

Discussion

7.1 Conclusion and Summary

Network Representation

We have implemnted a new representation of the network that is suited equally well for multithreaded and distribution simulation of neuronal networks. The concept of virtual processes provides an elegant way to ensure reproducible results when the number of virtual processes is kept constant. At the same time it increases the performance of the system, in particular its scalability, considerably in contrast to the previous version of NEST because the memory used by different threads is now separated and thus cache thrashing is reduced.

Distributed Simulation and Network Construction

The availability of distributed simulation provides decisive advantages over the multithreaded simulation scheme. It enables researchers to simulate networks far exceeding the memory available on a single computer and exploit the computing power of computer clusters. Simulataneously this technique reduces network construction time because the network can be built in parallel.

Connection System

The new connection system uses synapse prototypes, which provide flexible ways to implement synaptic plasticity and build heterogeneous networks. The framework has a convenient SLI interface to manage default values for the different types of synapses and change the parameters of connections.

Performance

The performance of the new simulation kernel is now comparable to Paranel's. Due to the new network representation and the new scheduler, the ring buffers of the nodes do not have to be thread safe anymore. This lowers the risk of cache thrashing and leads to better overall performance of the application.

7.2 Critique

Fixed Distribution of Nodes

The nodes are distributed onto the threads and processes by using a hardcoded modulo algorithm. This is certainly the best approach for the general case, but can restrict the flexibility for certain networks. For example if structured networks are simulated, the activity is mainly localized to certain areas, that one would like to put on a single computer to minimize the network traffic.

Equal Number of Thread in all Processes

Currently, the scheduling algorithm requires that each process has to run the same number of threads. This makes the application of NEST2 on heterogeneous computer clusters mostly impossible.

Large Number of Proxy Nodes

If a large number of processes is used, like e.g. on large computer clusters Markram (e.g. 2006), the memory requirements due to the proxy nodes may become high. A solution to this problem has already been given in chapter 2: A lookup-table has to be stored in each virtual process that can be used to address the nodes.

Scaling of NEST2 with Multithreading

Although the performance of the new kernel has been greatly improved with respect to scalability, one question still remains: why does the multithreaded version scale worse than the message-passing variant? The gain in performance due to the improvements of the memory management suggests that cache utilization and memory layout are indeed the main problems here. Due to the multithreaded approach, a complete separation of memory by thread is impossible. However, the objects that consume most memory are known. The following list contains the objects together with the state of memory separation for this kind of object:

- Nodes: the memory of the nodes is separated by the pool allocator.
- Ring buffers: the buffers are allocated in order of threads, and thus separated
- Connections: the connections are allocated in order of creation and not separate. A solution could be to use the pool allocator also for the `Connector` objects.

To find other candidates for further optimizations, the only way will be to generate a detailed profile of NEST's run time, which will allow an analysis of the time consumption of the different functions.

Restrictions with Distributed Simulation

Albeit the distributed simulation scheme allows for greater flexibility, it also has some drawbacks in the current implementation. Only spikes can be sent to remote targets, which limits the communication between the elements. However, at the moment only spiking neuron models

are implemented in NEST. Another problem of the distributed simulation setup is that it only allows script-driven execution of batch jobs, whereas the multithreaded program can also be used interactively, which is a convenient way of setting up networks and experiment with different parameters. A possible solution for this is provided in the next section.

7.3 Outlook

Multithreaded Connection Setup

In chapter 4 we pointed out that the time for connection setup of a neuronal system can easily exceed the simulation time. This is a problem especially for very large networks. In Paranel the problem can be solved by using multiple processes that connect and simulate the network in parallel. This is also possible in NEST2. However, parallel connection establishment in a pure multithreaded simulation setup is not possible in the current implementation. This problem can be solved by using multiple threads to connect the network by using algorithms similar to those of the distributed case.

Serial Simulation of Multiple Virtual Processes

To enhance the possibility to obtain reproducible results, a new scheduler could be be implemented that allows the serial simulation of multiple virtual processes without the need to use several threads or processes.

Make Use of the new Connection Framework

In Paranel a lot of different synapse types are available for the simulation of heterogeneous networks. The new connection system of NEST allows the same flexibility as Paranel but only a static type of connection has been implemented so far. The next step will be to port the synapse types from Paranel to the new connection framework of NEST2.

Distributed SLI

As pointed out earlier, the interpreter supports two modes of operation: interactive at the command line prompt or script driven as virtual machine (see figure 7.1). A distributed setup could, in principle, use the same modes, with the difference that the interactive mode has to be mediated by a local process, the *master*, which takes part in the simulation and relays the commands to the other processes, the *slaves*. In the current implementation of NEST2, the master-slave mode is not implemented yet. The system has, however, been designed to make an implementation at a later time easy. Currently, only the script driven mode is available for distributed simulation setups.

Refined Communication Control

In the current implementation the events are broadcasted to every process in the simulation, regardless if the node has targets on the receiving process. A solution to this problem is already implemented in Paranel, where a list of the machines that actually have targets is stored for

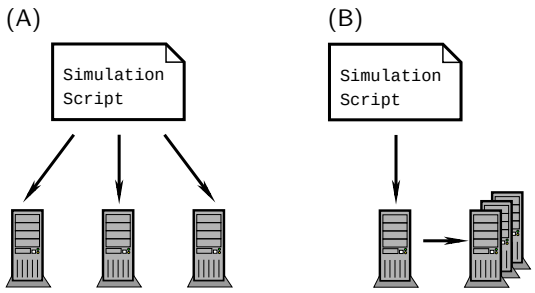


Figure 7.1: The interpreter in a distributed environment: (A) Non-interactive, in batch mode; Every computer executes only the commands that are relevant for it. (B) Interactive, in master-slave mode; The master application processes the commands interactively and sends the commands to the slaves for processing.

each node. This is particularly important for network sizes that go beyound the local networks within a qubic millimeter of cortex because then the probality for a neuron to have targets on every machine is smaller.

Bibliography

Adobe Systems Inc. (1991). *The PostScript Language Reference Manual* (2 ed.). Addison-Wesley.

Aho, A. V., Sethi, R., & Ullman, J. D. (1988). *Compilers, principles, techniques, and tools*. Reading, Massachusetts: Addison-Wesley.

Bower, J. M., & Beeman, D. (1997). *The Book of GENESIS: Exploring realistic neural models with the GEneral NEural SImulation System* (2 ed.). New York: TELOS, Springer-Verlag-Verlag.

Braitenberg, V., & Schüz, A. (1998). *Cortex: Statistics and Geometry of Neuronal Connectivity* (2nd ed.). Berlin: Springer-Verlag-Verlag.

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci. 8*(3), 183–208.

Carnevale, T., & Hines, M. (2006). *The NEURON Book*. Cambridge: Cambridge University Press.

Diesmann, M., & Gewaltig, M.-O. (2002). NEST: An environment for neural systems simulations. In T. Plesser & V. Macho (Eds.), *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, Volume 58 of *GWDG-Bericht*, pp. 43–70. Göttingen: Ges. für Wiss. Datenverarbeitung.

Diesmann, M., Gewaltig, M.-O., & Aertsen, A. (1995). SYNOD: an environment for neural systems simulations. Language interface and tutorial. Technical Report GC-AA-/95-3, Weizmann Institute of Science, The Grodetsky Center for Research of Higher Brain Functions, Israel.

Finkel, R. A. (1996). *Advanced programming languages*. Menlo Park, California: Addison-Wesley.

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3 ed.). Addison-Wesley Professional.

Fregnac, Y. (1998). Homeostasis or synaptic plasticity? *Nature 391*, 845–846.

Fujimoto, R. M. (2000). *Parallel and distributed simulation systems*. New York: Wiley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesely.

Gewaltig, M.-O., & Diesmann, M. (2006). Exploring large-scale models of neural systems with the neural simulation tool nest. Contribution to CNS 2006.

Gross, J., & Yellen, J. (1999). *Graph Theory and its Applications*. CRC Press.

Hodgkin, A. L., & Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol. (Lond) 117*, 500–544.

Larkum, M. E., Zhu, J. J., & Sakmann, B. (2001). Dendritic mechanisms underlying the coupling of the dendritic with the axonal action potential initiation zone of adult rat layer 5 pyramidal neurons. *J. Physiol. (Lond) 533.2*, 447–466.

Lewis, B., & Berg, D. J. (1997). *Multithreaded Programming With PThreads*. Sun Microsystems Press.

Lutz, M. (2001). *Programming Python, Second Edition with CD*. O'Reilly.

MacGregor, R. J. (1987). *Neural and Brain Modeling*. San Diego: Academic Press.

Markram, H. (2006). The blue brain project. *Nat. Rev. Neurosci. 7*, 153–160.

MathWorks (2002). *MATLAB The Language of Technical Computing: Using MATLAB*. Natick, MA. 3 Apple Hill Drive, Natick, Mass. 01760-2098.

Message Passing Interface Forum (1994). MPI: A message-passing interface standard. Technical Report UT-CS-94-230.

Morrison, A., Aertsen, A., & Diesmann, M. (2005). Spike-timing dependent plasticity in balanced random networks. *submitted*.

Morrison, A., & Diesmann, M. (2003). Cache effects in distributed simulation of biological neural networks. Internal report.

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., & Diesmann, M. (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput. 17*(8), 1776–1801.

Morrison, A., Straube, S., Plesser, H. E., & Diesmann, M. (2005). Exact subthreshold integration with continuous spike times in discrete time neural network simulations. *submitted*.

Research Systems Inc. (1987). Interactive Data Language. http://www.rsinc.com/idl/.

Stroustrup, B. (1997). *The C++ Programming Language* (3 ed.). New York: Addison-Wesely.

Tam, A., & Wang, C. (2000). Efficient scheduling of complete exchange on clusters. In *13th International Conference on Parallel and Distributed Computing Systems (PDCS 2000)*, Las Vegas.

Tanenbaum, A. S. (1999). *Structured Computer Organization* (4 ed.). Upper Saddle River: Prentice Hall.

Tsodyks, M., Pawelzik, K., & Markram, H. (1998). Neural networks with dynamic synapses. *Neural Comput. 10*, 821–835.

Tsodyks, M., Uziel, A., & Markram, H. (2000). Synchrony generation in recurrent networks with frequency-dependent synapses. *J. Neurosci. 20*, RC1 (1–5).

Tuckwell, H. C. (1988). *Introduction to Theoretical Neurobiology*, Volume 1, Chapter 3, The Lapique model of the nerve cell, pp. 85–123. Cambridge: Cambridge University Press.

Wilkinson, B., & Allen, M. (2004). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers* (2 ed.). Prentice Hall.

Wolfram, S. (2003). *The Mathematica Book* (5 ed.). Wolfram Media Incorporated.

Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000). *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems* (2 ed.). Amsterdam: Academic Press.