Supercomputational Neuroscience: Complex Networks in Brain Dynamics

Peter beim Graben, Changsong Zhou, Marco Thiel, and Jürgen Kurths (Eds.)

Institute of Physics, University of Potsdam, Germany

Contents

Part I

Neurophysiology

Part II

Complex Networks

Part III

Cognition and Higher Perception

Part IV

Implementations

1

Sequential and Parallel Implementation of Networks

Werner von Bloh

Potsdam Institute for Climate Impact Research,
PO Box 601203, 14412 Potsdam, Germany,
bloh@pik-potsdam.de

1.1 Implementation of Diffusive Coupled Networks

A diffusive coupled network of n neurons can be described by a state vector $N_i(t)$:

$$\frac{\mathrm{d}N_i(t)}{\mathrm{d}t} = f(N_i(t)) + \sum_{j=1}^{n} w_{ij} \times (N_j(t) - N_i(t)) \,, \tag{1.1}$$

where the function f describes the evolution of the neuron and the weight matrix w_{ij} the coupling strength between the neurons. For simplicity, we assume that the state of the neuron can be described by a single scalar. The matrix w_{ij} has the following properties:

$$w_{ij} \begin{cases} > 0 \text{ excitatory coupling} \\ = 0 \text{ no coupling} \\ < 0 \text{ inhibitory coupling} \end{cases} \tag{1.2}$$

The matrix can be implemented by a two-dimensional array. The size of this array scales with $O(n^2)$ independent of the number of couplings (equivalent to non-zero elements in w_{ij}). This is in particular inefficient for sparsely coupled neurons. In this case, it is better to use the list structure (see Fig. 1.1) [1].

From now, on we will apply the Morris-Lecar (ML) neuron model [2] (see Chapts. I, V, V). The dynamics can be described by two coupled ordinary differential equations

(a)

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

(b)

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

Fig. 1.1. (a) Adjacency matrix and (b) list representation of a random network of 16 neurons with connection probability of $p = 0.15$

$$
\begin{aligned}
c\frac{\mathrm{d}v}{\mathrm{d}t} &= f_v(v,w) = I - g_L(v - v_L) - g_\mathrm{K} w(v - v_\mathrm{K}) \\
&\quad - g_\mathrm{Ca} m_v(v)(v - v_\mathrm{Ca}) \qquad (1.3) \\
\frac{\mathrm{d}w}{\mathrm{d}t} &= f_w(v,w) = \lambda(v)(w_v(v) - w)\,, \qquad (1.4) \\
m_v(v) &= \frac{1}{2}(1 + \tanh((v - v_1)/v_2))\,, \\
m_w(v) &= \frac{1}{2}(1 + \tanh((v - v_3)/v_4))\,, \\
\lambda(v) &= \theta \cosh((v - v_3)/(2v_4))\,.
\end{aligned}
$$

The diffusive coupling (corresponding to electrical synapses; cf. Chapts. I and IV) is achieved with the component v, i.e

$$
c\frac{\mathrm{d}v_i}{\mathrm{d}t} = f_v(v_i, w_i) + \sum_{j=1}^{n} w_{ij} \times (v_j - v_i). \qquad (1.5)
$$

1.1.1 Sequential Code

The implementation of the list and the dynamics of the network is performed with the help of the programming language C. It allows the definition of abstract datatypes, where definition and implementation are separate [3]. An introduction to the programming language C is given in [4]. A neuron can be declared as a datatype `Neuron` in the following way (in the declaration header file `neuron.h`):

```
typedef struct
{
  float v[2];   /* state of ML neuron */
  int inhib;    /* inhibitory (TRUE) or not (FALSE) */
  List connect; /* list of connections */
} Neuron;
```

The datatype of lists `List` used by `Neuron` is declared in the header file `list.h`:

```
typedef struct
{
  int *index; /* pointer to list of indices itself */
  int len;    /* length of list */
} List;

/* Declaration of functions */

/* initialize empty list to  */
extern void initlist(List *);
/* add connection to list */
extern int  addlistitem(List *,int);
```

The header file contains the declaration of the structure `List` and the prototypes of the functions for initialization and adding a connection to the network. Lists can be implemented in two ways: (1) as a variable sized array, and (2) with linked pointers. The advantage of a implementation using arrays is the better storage efficiency. A pointer implementation uses for every element an additional pointer causing some memory overhead. Adding an element to an array, however, is performed by a copying process of the full list not necessary in the pointer implementation. If the network is not changed often during runtime, an array implementation is usually more efficient. The initialization function sets the length of the list to zero and initializes the index array to the `NULL` pointer.

```
void initlist(List *list)
{
  list->len=0;
  list->index=NULL;
} /* of 'newlist' */
```

An item can be added to the list by a call to the `addlistitem` function. The updated length of the list is returned by this function. The memory allocation can be done by the `realloc` function of the standard C library. It increases the allocated memory by a certain number of bytes.

```
int addlistitem(List *list,int item)
{
  /* add item to index vector */
```

```
  list->index=(int *)realloc(list->index,
                             sizeof(int)*(list->len+1));
  list->index[list->len]=item;
  list->len++;
  return list->len;
} /* of 'addlistitem' */
```

A random network can be built up with the function `randomnet`. For each element of the neuron array, the list is first initialized and then random connections are added to the list with a probability p_{conn}. The neuron is marked as an inhibitory neuron with a probability p_{inhib}.

```
void randomnet(Neuron net[], /* array of neurons */
               int n,        /* number of neurons */
               float p_conn,  /* probability of establishing
                                 connection */
               float p_inhib /* probability of a inhibitory
                                neuron */
              )
{
  int i,j;
  for(i=0;i<n;i++) /* iterate over all neurons */
  {
   initlist(&net[i].connect);
   net[i].inhib=(drand48()<p_inhib);
   for(j=0;j<n;j++)
     if(i!=j &&  /* avoid self connections */
        drand48()<p_conn)
       addlistitem(&net[i].connect,j);
  }
} /* of 'randomnet' */
```

The `drand48` function is a generator of uniformly distributed pseudo-random numbers defined in `stdlib.h`. The update of the ML model can be implemented by the following `updateml` function:

```
/* constants for ML model */
#define   c 1.0
#define   gL 0.5
#define   gK 2.0
#define   gCa 1.0
#define   vL (-0.5)
#define   vK (-0.7)
#define   vCa 1.0
#define   v1 (-0.01)
#define   v2 0.15
#define   v3 0.1
```

```
#define   v4 0.145
#define   theta (1.0/3.0)
void updateml(float dv[2], /* derivatives dv/dt  */
              float v[2],  /* state vector v, v[0]=v, v[1}=w */
              float I      /* applied current */
             )
{
  float mv,wv,lambda;
  mv=0.5*(1+tanh((v[0]-v1)/v2));
  wv=0.5*(1+tanh((v[0]-v3)/v4));
  lambda=theta*cosh((v[0]-v3)/(2*v4));
  dv[0]=I-gL*(v[0]-vL)-gK*v[1]*(v[0]-vK)-gCa*mv*(v[0]-vCa);
  dv[1]=lambda*(wv-v[1]);
} /* of 'updateml' */
```

Then the update of the state of all coupled neurons can be performed by the `update` function. A simple explicit Euler scheme is used to solve the ordinary differential equations:

$$\begin{aligned} v_i(t+\Delta t) &= v_i(t) + \frac{\Delta t}{c} \times f_v(v_i(t), w_i(t)), \\ w_i(t+\Delta t) &= w_i(t) + \Delta t \times f_w(v_i(t), w_i(t)). \end{aligned} \tag{1.6}$$

```
void update(Neuron net_new[], /* updated array of neurons */
            Neuron net[],     /* array of neurons */
            int n,            /* number of neurons */
            float I,          /* applied current */
            float w_in,       /* inhibitory coupling strength */
            float w_ex,       /* excitatory coupling strength */
            float h           /* time step */
           )
{
  int i,j,index;
  float sum,dv[2];
  for(i=0;i<n;i++) /* calculate coupling */
  {
    sum=0;
    for(j=0;j<net[i].connect.len;j++)
    {
       index=net[i].connect.index[j];
       if(net[index].inhib)
          /* inhibitory neuron */
          sum-=w_in*(net[index].v[0]-net[i].v[0]);
        else
          /* excitatory neuron */
          sum+=w_ex*(net[index].v[0]-net[i].v[0]);
```

```
      }
      updateml(dv,net[i].v,I);
      /* apply simple Euler scheme */
      net_new[i].v[0]=net[i].v[0]+(h/c)*(dv[0]+sum);
      net_new[i].v[1]=net[i].v[1]+h*dv[1];
  } /* of 'update' */
```

1.1.2 Parallel Code

The sequential code is limited by the speed and storage capacity of a single workstation. In particular, a large network of neurons uses $O(n^2)$ memory for the connections, quite easily exceeding the storage of a typical workstation. Instead of running the code on a faster computer with more memory, it is usually more cost efficient to run the code in parallel on a network of computers. There are two different parallel computational models: shared memory and distributed memory. In the shared memory model, the processors share the same memory and address space. The memory bandwidth, however, limits the achievable maximum number of processors. In the distributed memory model, each processor has its own local memory. Data exchange is performed via a communication network. This approach allows parallel computers consisting of several thousands of processors.

Message-passing paradigm

The parallelization of the code uses the message-passing paradigm based on the distributed memory model. The message-passing model consists of a set of processes or tasks that only have local memory but are able to communicate with other tasks by sending and receiving messages. The data transfer from the local memory of one task to the local memory of another task requires operations to be performed on both processes. A portable implementation running on different platforms and architectures is provided by the message-passing interface (MPI). An introduction to MPI is given in [5]. The basic

Table 1.1. The basic six-function version of MPI

Routine	Description
`MPI_Init`	Initialize MPI
`MPI_Comm_size`	Find out how many tasks there are
`MPI_Comm_rank`	Find out which task I am
`MPI_Finalize`	Finish MPI
`MPI_Send`	Send a message
`MPI_Recv`	Receive a message

functions of MPI are listed in Table 1.1. The datatypes and message-passing

functions are declared in the header `mpi.h`: A simple *hello world* program in MPI looks like:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc,char **argv)
{
  int mytask,ntask;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&ntask);
  MPI_Comm_rank(MPI_COMM_WORLD,&mytask);
  printf("Hello! I am task %d out of %d tasks\n",
         mytask,ntask);
  MPI_Finalize();
  return 0;
}  /* of 'main' */
```

This code can be run in parallel on a arbitrary number of processors. On four processors, it will produce the following output:

```
% mpirun -np hello
Hello! I am task 0 out of 4 tasks
Hello! I am task 2 out of 4 tasks
Hello! I am task 1 out of 4 tasks
Hello! I am task 3 out of 4 tasks
```

The task belonging to a group of n tasks is identified by a number ranging from 0 to $n-1$. The default group containing all tasks is named `MPI_COMM_WORLD`. There are two types of communication routines. The first class are point to point routines, like `MPI_Recv` and `MPI_Send` in order to send to/receive from a specified task (Fig 1.2). The basic send operation in MPI is declared as

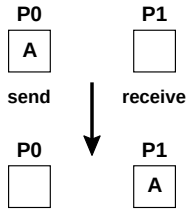

Fig. 1.2. Basic message-passing function of sending information from task P0 to task P1

```
MPI_Send(address,count,datatype,destination,tag,comm),
```

where

- `address, count, datatype` describe `count` occurrences of items of the form `datatype` starting at `address`.
- `destination` is the task identifier of the destination in the group associated with the communicator `comm`.
- `tag` is an integer used for message matching.
- `comm` identifies a group of tasks and a communication context.

The corresponding receive is

```
MPI_Recv(address,maxcount,datatype,source,tag,comm,status)
```

where

- `address, maxcount, datatype` describe the receive buffer as they do in the case of `MPI_Send`.
- `source` is the task identifier of the source of the message in the group associated with the communicator `comm`.
- `status` holds information about the actual message size, source and tag.

MPI has predefined datatypes of the objects sent to or received from remote tasks. They are listed in Table 1.2. The size of the MPI datatypes can be calculated by a call of the `MPI_Type_extent` routine.

Table 1.2. Subset of basic (predefined) MPI datatypes in C

MPI Datatype	C Datatype
`MPI_BYTE`	`signed char`
`MPI_DOUBLE`	`double`
`MPI_FLOAT`	`float`
`MPI_INT`	`int`

The second class of MPI functions are collective operations that are called by all processors simultaneously. The most important collective routines are summarized in Table 1.3. Their communication patterns are graphically represented in Fig. 1.3. Finally, the C bindings of all MPI routines used are given in Table 1.4.

Table 1.3. Collective operations of the MPI

Routine	Description
`MPI_Bcast`	Broadcast data to all tasks
`MPI_Gather`	Gather all data to a single task
`MPI_Reduce`	Reduce data to one task
`MPI_Alltoall`	All to all communication
`MPI_Alltoallv`	All to all communication with variable size
`MPI_Scatter`	Scatter data to all tasks

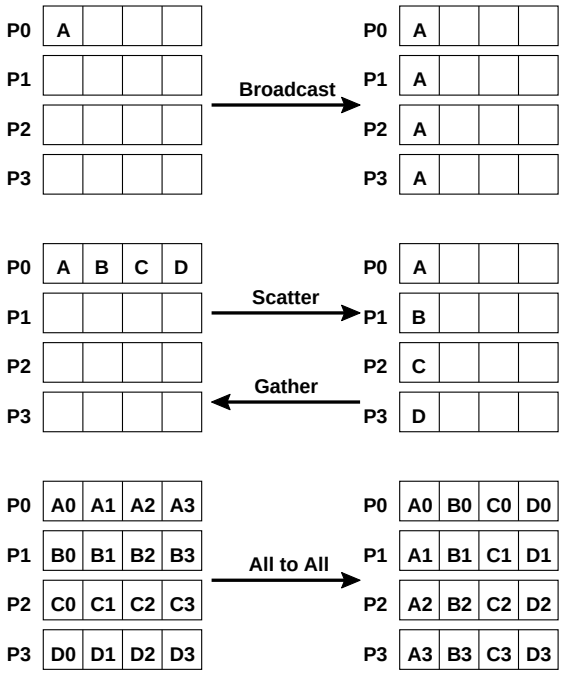

Fig. 1.3. Communication patterns for the MPI collective operations `MPI_Bcast`, `MPI_Gather`/`MPI_Scatter`, and `MPI_Alltoall` on four tasks P0–P3

Distributing the network

In a parallel application, the neurons have to be distributed evenly on all tasks. The parallel algorithm is described in [6] in detail (cf. Chapt. IV). They use basic send/receive routines, while our implementation is based on collective MPI operations. An algorithm based on send/receive must use a complete pairwise exchange algorithm [7] in order to prevent deadlocks.

In a distributed network, the connection list contains entries to remote neurons. Then, the state of the neuron has to be transferred to the remote neuron stored in a different task. The algorithm for setting up the communication structure works in the following way:

1. For each task, a sorted list of neuron indices to which a connection exists has to be created. Sorting is necessary in order to delete duplicate entries.
2. It has to be determined how many neuron states have to be sent to remote neurons of all tasks. This defines the length of the output buffers.
3. This information is distributed to all tasks via a `MPI_Alltoall` collective operation call.

Table 1.4. C bindings for the MPI functions used in the parallelization of networks

int **MPI_Alltoall**(void *sendbuf,int sendcount,MPI_Datatype sendtype, void *recvbuf,int recvcount,MPI_Datatype recvtype, MPI_Comm comm)
int **MPI_Alltoallv**(void *sendbuf,int *sendcounts,int *sdispls, MPI_Datatype sendtype,void *recvbuf,int *recvcounts, int *rdispls,MPI_Datatype recvtype,MPI_Comm comm)
int **MPI_Comm_size**(MPI_Comm comm,int *size)
int **MPI_Comm_rank**(MPI_Comm comm,int *rank)
int **MPI_Finalize**()
int **MPI_Gather**(void *sendbuf,int sendcount,MPI_Datatype sendtype, void *recvbuf,int recvcount,MPI_Datatype recvtype,int root, MPI_Comm comm)
int **MPI_Init**(int *argc,char ***argv)
int **MPI_Reduce**(void *sendbuf,void *recvbuf,int count,MPI_Op op,int root, MPI_Comm comm)
int **MPI_Recv**(void *buf,int count,MPI_Datatype datatype,int source,int tag, MPI_Comm comm)
int **MPI_Send**(void *buf,int count,MPI_Datatype datatype,int source,int tag, MPI_Comm comm,MPI_Status *status)
int **MPI_Type_extent**(MPI_datatype datatype,MPI_Aint *extent)

4. The indices of the neurons that have to be sent to a specific task must be distributed from all tasks to all tasks via a `MPI_Alltoallv` call. The length of the packets has been determined by step 2 and 3.
5. Two index vectors are built up containing the mapping from the input buffer to the connection lists and the neuron indices to the output buffer.

Then, the exchange of the neuron states can be performed in three steps:

1. Copy the state of the neurons to be sent to other tasks to the output buffer.
2. Distribute the information with a call of `MPI_Alltoallv`. The `MPI_Alltoallv` collective operation transports the output buffers to the corresponding input buffers.
3. The input buffer must be mapped to the connection list.

The datatype of a distributed network `Pnet` can be defined by the following structure:

```
typedef struct
{
  int n;  /* total number of neurons */
  int lo; /* lower bound of subarray */
  int hi; /* upper bound of subarray */
  int ntask; /* number of tasks */
  int taskid;  /* my task identifier */
```

```
  int outsize; /* size of output buffer */
  int insize; /* size of input buffer */
  int *outdisp; /* displacement vector  for output */
  int *indisp; /* displacement vector for input */
  int *inlen,*outlen; /* vector length for input/output */
  MPI_Datatype datatype; /* datatype  of input/output buffer */
  void *outbuffer,*inbuffer; /* input/output buffer of
                             /* generic type void */
  int *outindex; /* index vector of output */
  List *connect;  /* list of connections */
} Pnet;
```

The topology of the network is now part of this datatype and not part of `neuron`. The basic functions for the datatype `Pnet` are:

```
/* Initialization of datatype */
extern Pnet *pnet_init(MPI_Datatype,int);
/* Random network setup */
extern void pnet_random(Pnet *,float);
/* Creating communication structure */
extern void pnet_setup(Pnet *);
/* Exchange information */
extern void pnet_exchg(Pnet *);
/* Macros for convenience */
/* iterator of subarray */
#define pnet_foreach(pnet,i) for(i=(pnet)->lo;i<=(pnet)->hi;i++)
/* allocating an array ar[lo:hi] of datatype type */
#define newvec(type,lo,hi) \
   (type *)malloc(sizeof(type)*(hi-(lo)+1)-(lo)
#define freevec(ptr,lo) free(ptr+(lo))
```

The datatype is initialized by a call to `pnet_init`. The function has two arguments. The first argument defines the datatype of the data to be distributed between the different tasks. The second argument defines the total number of neurons. In the first part of the function, the necessary arrays are allocated, the number of tasks and the task identifier are determined. In the next part, the lower and upper bounds of the neuron array are calculated. In particular, it must be considered that the total number of neurons cannot be divided by the total number of tasks. Finally, an array for the neuron connections is allocated and initialized:

```
Pnet *pnet_init(MPI_Datatype datatype, /* MPI datatype */
                int n /* total number of neurons */
               )      /* returns allocated struct */
{
  int slice,rem,i;
  Pnet *pnet;
```

```
  pnet=(Pnet *)malloc(sizeof(Pnet));
  pnet->n=n;
  pnet->datatype=datatype;
  MPI_Comm_size(MPI_COMM_WORLD,&pnet->ntask);
  MPI_Comm_rank(MPI_COMM_WORLD,&pnet->taskid);
  /* calculate lower and upper bound of subarray */
  slice=pnet->n/pnet->ntask;
  pnet->lo=pnet->taskid*slice;
  pnet->hi=(pnet->taskid+1)*slice-1;
  rem=pnet->n % pnet->ntask;
  /* distribute the remainder evenly on all tasks */
  if(pnet->taskid<rem)
  {
    pnet->lo+=pnet->taskid;
    pnet->hi+=pnet->taskid+1;
  }
  else
  {
    pnet->lo+=rem;
    pnet->hi+=rem;
  }
  /* allocate arrays */
  pnet->outdisp=(int *)malloc(sizeof(int)*pnet->ntask);
  pnet->indisp=(int *)malloc(sizeof(int)*pnet->ntask);
  pnet->outlen=(int *)malloc(sizeof(int)*pnet->ntask);
  pnet->inlen=(int *)malloc(sizeof(int)*pnet->ntask);
  pnet->connect=newvec(List,pnet->lo,pnet->hi);
  pnet_foreach(pnet,i)
    initlist(pnet->connect+i);
  return pnet;
} /* of 'pnet_init' */
```

A random network is set up by calling `pnet_random`. The function uses the macro `pnet_foreach`, ensuring that only the local subarray of each task is accessed.

```
void pnet_random(Pnet *pnet,
                 float p_conn /* connection probability */
                )
{
  int i,j;
  pnet_foreach(pnet,i)
    for(j=0;j<pnet->n;j++)
      if(i!=j && drand48()<p_conn)
        addlistitem(pnet->connect+i,j);
} /* of 'pnet_random' */
```

The setup of the necessary communication patterns between the different tasks is performed by the `pnet_setup` function. Each task has to know the upper and lower bounds of the subarrays of all other tasks. This information is stored in the arrays `lo` and `hi`:

```
void pnet_setup(Pnet *pnet)
{
  int *lo,*hi;
  int i,j,k,*in,size,slice,rem,task;
  slice=pnet->n/pnet->ntask;
  rem=pnet->n % pnet->ntask;
  lo=newvec(int,0,pnet->ntask-1);
  hi=newvec(int,0,pnet->ntask-1);
  for(i=0;i<pnet->ntask;i++)
  /* calculate boundaries of all tasks for n mod ntask<>0 */
  {
    lo[i]=i*slice;
    hi[i]=(i+1)*slice-1;
    if(i<rem)
    {
       lo[i]+=i;
       hi[i]+=i+1;
    }
    else
    {
      lo[i]+=rem;
      hi[i]+=rem;
    }
  }
```

Then, the total number of connections and their indices are calculated. The array `in` stores the neuron indices of all connections (Fig. 1.4).

```
  for(i=0;i<pnet->ntask;i++)
    pnet->outlen[i]=pnet->inlen[i]=0;
  /* calculating total length of connection list */
  size=0;
  pnet_foreach(pnet,i)
    size+=pnet->connect[i].len;
  in=(int *)malloc(sizeof(int)*size);
  k=0; /* concatenating connection lists */
  pnet_foreach(pnet,i)
    for(j=0;j<pnet->connect[i].len;j++)
      in[k++]=pnet->connect[i].index[j];
```

This array is sorted and duplicated entries are deleted (see Fig. 1.5).

```
  /* sort connection list */
```

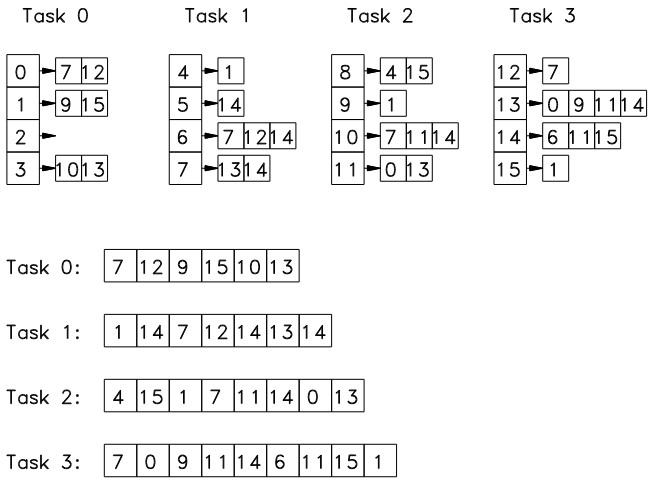

Fig. 1.4. Network of 16 neurons distributed to 4 tasks (upper figure). These are the contents of `pnet->connect` before the call of `pnet_setup`. The lower figure shows the combined lists of network connections for each task 0–3.

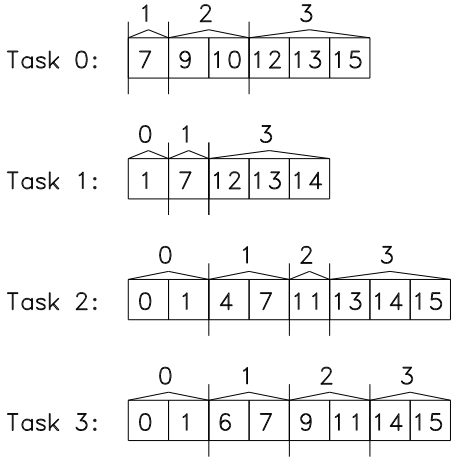

Fig. 1.5. List of network connections for each task after sorting and deleting duplicate entries. The numbers above the lists are the corresponding task identifiers to which the information must be sent.

```
qsort(in,size,sizeof(int),
      (int (*)(const void *,const void *))compare);
pnet->insize=1; /* delete duplicate entries */
for(i=1;i<size;i++)
  if(in[i]!=in[i-1]) /* same indices? */
  {
    /* no, increase insize by one */
    in[pnet->insize]=in[i];
    pnet->insize++;
  }
```

After this step, the length of the vector sent to all tasks has to be calculated. This can be performed by counting all neuron indices inside the lower and upper bound for each task (Fig. 1.6).

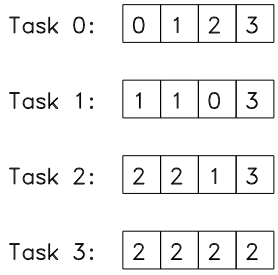

Fig. 1.6. The length of the communication packets needed by `MPI_Alltoallv`. The information has to be distributed by `MPI_Alltoall` to all tasks.

```
task=0;
pnet->inlen[task]=0;
/* calculating inlen vector */
for(i=0;i<pnet->insize;)
  if(in[i]<=hi[task]) /* inside the boundaries of task? */
  {
    /* yes, increment inlen by one */
    pnet->inlen[task]++;
    i++;
  }
  else
  {
    /* no, goto next task, set new inlen to zero */
    task++;
    pnet->inlen[task]=0;
  }
```

It is important to map the connection from the input buffer to the connection list of the local neurons. This information is stored again in the array of lists `pnet->connect`, because the original connection lists are not needed any more (Fig. 1.7). The current implementation uses a linear search algorithm to find the indices in the connection list leading to a runtime characteristic of $O(n^2)$. By using a better algorithm (e.g., binary search), the runtime can be significantly reduced. The setup function, however, is called only once during initialization of the network.

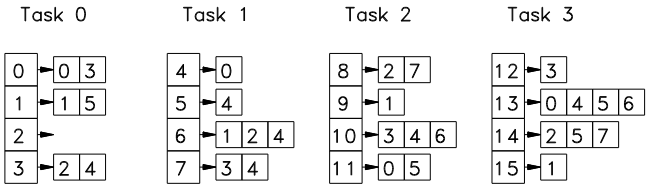

Fig. 1.7. Mapping of the input buffer to the connection list. These are the contents of `pnet->connect` after the call of `pnet_setup`.

```
/* calculating mapping from input buffer to connection */
pnet_foreach(pnet,i)
{
  for(j=0;j<pnet->connect[i].len;j++)
    /* search for index in array in */
    for(k=0;k<pnet->insize;k++)
      if(in[k]==pnet->connect[i].index[j])
      {
        /* index found and stop searching */
        pnet->connect[i].index[i]=k;
        break;
      }
} /* of pnet_foreach */
```

The information of the lengths of the communication packets to be received from other tasks has to be distributed by a call to `MPI_Alltoall`. Then, the `pnet->outlen` array contains the length of the outgoing packets. This information is needed by the `MPI_Alltoallv` function. `MPI_Alltoallv` sends a distinct message from each task to every task, where the messages can have different sizes and displacements. The displacement is the offset from the first element of the array to the first element of the message and can be simply calculated by summing up the `pnet->inlen` array. After calling `MPI_Alltoallv`, the array `pnet->outindex` contains the indices of neurons that must be sent to other tasks (see Fig. 1.8).

```
MPI_Alltoall(pnet->inlen,1,MPI_INT,pnet->outlen,1,
```

```
              MPI_INT,MPI_COMM_WORLD);
/* calculating displacements needed by MPI_Alltoallv */
pnet->indisp[0]=pnet->outdisp[0]=0;
for(k=1;k<pnet->ntask;k++)
{
  pnet->indisp[k]=pnet->inlen[k-1]+pnet->indisp[k-1];
  pnet->outdisp[k]=pnet->outlen[k-1]+pnet->outdisp[k-1];
}
pnet->outsize=0;
for(i=0;i<pnet->ntask;i++)
  pnet->outsize+=pnet->outlen[i];
/* allocating outindex */
pnet->outindex=(int *)malloc(sizeof(int)*pnet->outsize);
/* information is moved from in to pnet->outindex calling
   the collective operation MPI_Alltoallv */
MPI_Alltoallv(in,pnet->inlen,pnet->indisp,MPI_INT,
              pnet->outindex,pnet->outlen,
              pnet->outdisp,MPI_INT,MPI_COMM_WORLD);
```

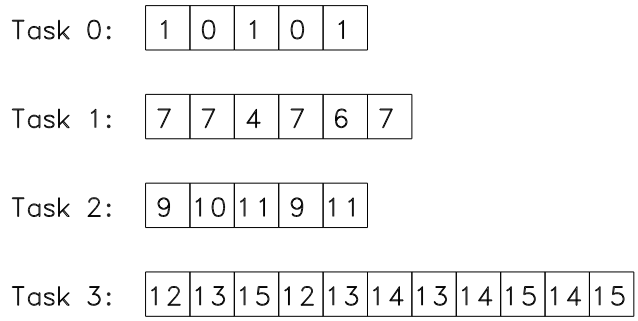

Fig. 1.8. `pnet->outindex` describes the mapping of the output buffer to the neuron indices.

Finally, we allocate the input and output buffers of type stored in `pnet->datatype`. The function `MPI_Type_extent` returns the number of bytes for this datatype. These buffers of generic type `void` have to be cast by the user to the appropriate type.

```
/* allocating input and output buffer */
MPI_Type_extent(pnet->datatype,&size);
pnet->outbuffer=malloc(pnet->outsize*size);
pnet->inbuffer=malloc(pnet->insize*size);
/* free auxiliary storage */
free(in);
free(lo);
```

```
  free(hi);
} /* of 'pnet_setup' */
```

The `compare` function is needed for the quicksort C library function `qsort`. It sorts integer values in ascending order:

```
static int compare(const int *a,const int *b)
{
  return *a-*b;
} /* of 'compare' */
```

Then, the exchange is done by the `pnet_exchg` function calling only the `MPI_Alltoallv` function. Information is copied from the output buffers to the input buffers.

```
void pnet_exchg(Pnet *pnet)
{
  MPI_Alltoallv(pnet->outbuffer,
                pnet->outlen,pnet->outdisp,
                pnet->datatype,pnet->inbuffer,
                pnet->inlen,pnet->indisp,
                pnet->datatype,MPI_COMM_WORLD);
} /* of 'pnet_exchg' */
```

The parallel initialization of the network is performed by `init`:

```
Neuron *init(Pnet **pnet,
             int n, /* number of neurons */
             float p_conn /* connection probability */
            ) /* returns allocated array of neurons */
{
  Neuron *net;
  int i;
  *pnet=pnet_init(MPI_FLOAT,n);
  pnet_random(*pnet,p_conn);
  pnet_setup(*pnet);
  net=newvec(Neuron,(*pnet)->lo,(*pnet)->hi);
  /* random initial state of ML neurons */
  pnet_foreach(*pnet,i)
  {
    net[i].v[0]=-0.02*0.01*drand48();
    net[i].v[1]=0.05+0.20*drand48();
  }
  return net;
} /* of 'init' */
```

Then, the parallel update for diffusive coupling can be done in the following way. For simplicity, only excitatory coupling is used. First, the state of the neuron has to be copied to the output buffers. After calling `pnet_exchg`, the

information is moved to the input buffer. The list `pnet->connect` provides the information about the mapping of the input buffer to the connections.

```
void update(Pnet *pnet,
            Neuron net[], /* subarray of neurons */
            float I,      /* applied current */
            float w,      /* coupling strength */
            float h       /* time step */
           )
{
  int i;
  float sum,*buffer,dv[2];
  /* cast outbuffer to float pointer */
  buffer=(float *)pnet->outbuffer;
  /* copy state to output buffer */
  for(i=0;i<pnet->outsize;i++)
    buffer[i]=net[pnet->outindex[i]].v[0];
  /* Exchange of necessary information to all tasks */
  pnet_exchg(pnet);
  /* cast inbuffer to float pointer */
  buffer=(float *)pnet->inbuffer;
  pnet_foreach(pnet,i)
  {
    sum=0;
    for(j=0;j<pnet->connect[i].len;j++)
      sum+=buffer[pnet->connect[i].index[j]]-net[i].v[0];
    updateml(dv,net[i].v,I);
    /* performing Euler step */
    net[i].v[0]+=(h/c)*(dv[0]+w*sum);
    net[i].v[1]+=h*dv[1];
  }
} /* of 'update' */
```

Efficiency of parallelization

Communication is necessary after every time step in the case of diffusive coupling. This limits the efficiency of the parallel code, in particular for large networks with dense couplings. Efficiency E is defined by

$$E(p) := \frac{T(1)}{p \times T(p)}\,, \tag{1.7}$$

where $T(p)$ denotes the computation time running on p parallel tasks. The computation time can be divided into two parts:

$$T = T_{\mathrm{calc}} + T_{\mathrm{comm}}\,, \tag{1.8}$$

where T_{calc} denotes the computation part and T_{comm} the communication part. In the case of a fully coupled network of n neurons, T_{comm} scales as $O(n^2)$, while the computation part scales as $O(n^2/p)$:

$$T(p) \sim \frac{n^2}{p} T'_{\text{calc}} + n^2 T'_{\text{comm}}, \tag{1.9}$$

Thus,

$$E(p) = \frac{T'_{\text{calc}}}{T'_{\text{calc}} + pT'_{\text{comm}}} \tag{1.10}$$

resembling Amdahl's law[8]. For large p, the total runtime is dominated by the communication time and the efficiency tends to zero.

1.2 Non-diffusive Coupling

The coupling of neurons can also be done in a different way: If a certain threshold is reached, a spike train is sent to remote neurons with a delay time t_{delay}. The spike can be parameterized by a gain function $g(t)$ with a rise time τ_1 and decay time τ_2:

Das stimmt leider so nicht: g beschreibt die postsynaptischen Potentiale. Daher mein Vorschlag in blau!

The coupling of neurons can also be done in a different way: If the integrated postsynaptic potentials (PSP) reach a certain threshold, a spike train is sent to remote neurons with a delay time t_{delay}. The PSP evoked by one single spike can be parameterized by a gain function $g(t)$ with a rise time τ_1 and decay time τ_2 (cf. Chapt. I):

$$g(t) = \frac{\exp(-t/\tau_1) - \exp(-t/\tau_2)}{\tau_1 - \tau_2} \tag{1.11}$$

For $t > t_{\max} \approx 15$ms and the chosen parameter $\tau_1 = 1$ ms and $\tau_2 = 2$ ms, the spike function g is nearly zero.

A spike is generated if v exceeds the critical threshold of zero at time t_{spike}. The coupled model of n neurons is defined by

$$\begin{aligned} c\frac{\mathrm{d}v_i}{\mathrm{d}t} &= f_v(v_i, w_i) + \sum_{j=1}^{n} \sum_{k=1}^{m_j} w_{\text{in,ex}} g(t - t_{\text{spike},j,k} - t_{\text{delay},i,j})(v_i - v_{\text{in,ex}}) \,, \\ \frac{\mathrm{d}w_i}{\mathrm{d}t} &= f_w(v_i, w_i) + \sigma\xi \,, \\ i &= 1, \ldots, n \,, \end{aligned} \tag{1.12}$$

where $t_{\text{spike},j,k}, k = 1, \ldots, m_j$ are the m_j spike times of neuron j, $w_{\text{in,ex}}$, $v_{\text{in,ex}}$ are weights for inhibitory/excitatory coupling, and ξ is additional Gaussian white noise with amplitude σ. The numerical discretization using an Euler scheme for a stochastic differential equation is:

$$\begin{aligned} v_i(t + \Delta t) &= v_(t) + \frac{\Delta t}{c} \times f_v(v_i(t), w_i(t)), \\ w_i(t + \Delta t) &= w_i(t) + \Delta t \times f_w(v_i(t), w_i(t)) + \sqrt{\Delta t} \times \xi. \end{aligned} \tag{1.13}$$

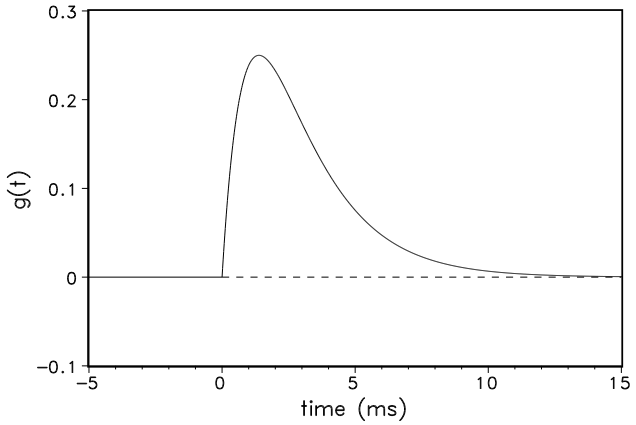

Fig. 1.9. PSP gain function $g(t)$ with rise time $\tau_1 = 1$ ms and decay time $\tau_2 = 2$ ms

1.2.1 Sequential Version

In order to speed up the evaluation of the gain function $g(t)$, a lookup table is used created by the function `getg`:

```
#define tau_1 1.0 /* rise time (ms) */
#define tau_2 2.0 /* decay time (ms) */

float *getg(int tmax, /* size of lookup table */
            float h  /* time step (ms) */
           )         /* returns calculated lookup table */
{
  float *g,gmax;
  int t;
  g=(float *)malloc(sizeof(float)*tmax);
  gmax=0;
  for(t=0;t<tmax;t++)
  {
   g[t]=1/(tau_1-tau_2)*(exp(-t*h/tau_1)-exp(-t*h/tau_2));
   if(g[t]>gmax)
     gmax=g[t];
  }
  /* normalize g */
  for(t=0;t<tmax;t++)
    g[t]/=gmax;
  return g;
} /* of 'getg' */
```

An efficient implementation of such coupling uses a finite event buffer where the time of spike events are stored (Chapt. IV). It is assumed that the max-

imum number of overlapping spikes is limited by the size of the buffer. The datatype `Rbuffer` can be defined by using an array that stores a limited number of events. After the maximum number of spikes is reached, the oldest event is overwritten. This can be implemented by a ring topology using the modulo operator. A graphical representation of datatype `Rbuffer` is shown in Fig. 1.10.

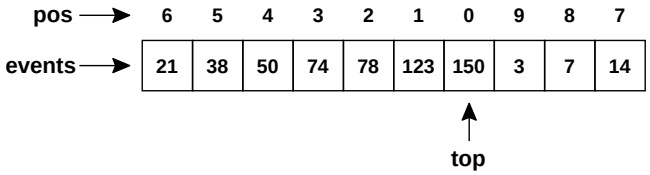

Fig. 1.10. Graphical representation of datatype `Rbuffer` with `size` equal to 10

```
typedef struct
{
  int len;  /* length of ring buffer */
  int top;  /* index of first element in ring buffer */
  int size; /* maximum length of ring buffer */
  int *events; /* stores up to size latest events */
} Rbuffer;
void initrbuffer(Rbuffer *,int);
void addrbuffer(Rbuffer *,int);
```

A macro is defined in order to access an event at position `pos` from the top of the ring buffer:

```
#define getrbuffer(rbuf,pos) \
  (rbuf)->events[((rbuf)->top-1-(pos)+(rbuf)->size) % (rbuf)->size]
```

The datatype is initialized by a call to `initrbuffer`. The parameter `size` sets the maximum number of events that can be stored.

```
void initrbuffer(Rbuffer *rbuf,int size)
{
  rbuf->len=rbuf->top=0;
  rbuf->events=(int *)malloc(sizeof(int)*size);
  rbuf->size=size;
} /* of 'initrbuffer' */
```

The implementation for adding an event to the ring buffer is straightforward. The modulo operator is used after incrementing the index to the first element in the buffer `top` defining a ring topology.

```
void addrbuffer(Rbuffer *rbuf,int event)
{
```

```
  /* have we reached the maximum number of events? */
  if(rbuf->len<rbuf->size)
    rbuf->len++; /* no, increase len by one */
  rbuf->events[rbuf->top]=event;
  /* increase top by one, then perform  modulo operator */
  rbuf->top=(rbuf->top+1) % rbuf->size;
} /* of 'addrbuffer' */
```

For the ML model, the datatype `Neuron` has to be modified:

```
typedef struct
{
  float v[2];    /* state of neuron (ML) */
  Rbuffer spikes;  /* times of last spikes */
  List connect;  /* list of connections */
  int inhib; /* inhibitory (TRUE) or excitatory (FALSE) */
} Neuron;
```

The neurons are initialized and the network is established by the function `init`:

```
#define RBUFF_LEN 10 /* maximum length of ring buffer */

void init(Neuron net[], /* array of neurons */
          int n,        /* total number of neurons */
          float p_conn, /* probability of establishing
                           connection */
          float p_inhib /* probability of a inhibitory
                           coupling */
         )
{
  int i,j;
  /* Setup of random network */
  randomnet(net,n,p_conn,p_inhib);
  for(i=0;i<n;i++)
  {
    /* Set initial conditions of neurons */
    net[i].v[0]=-0.02*0.01*drand48();
    net[i].v[1]=0.05+0.20*drand48();
    /* Initializing event ring buffer */
    initrbuffer(&net[i].spikes,RBUFF_LEN);
  }
} /* of 'init' */
```

The update function uses the datatype `Rbuffer` and `List`. The Gaussian white noise is generated by a call of the `gasdev` function as part of the numerical recipes library [9]. The function has been slightly modified and uses the `drand48` random number generator.

```
/* constants for ML model */
#define   c 1.0
/* coupling constants */
#define V_in (-0.55)
#define V_ex 0.05

void update(FILE *file, /* output file for spikes */
            Neuron net[], /* array of neurons */
            int n, /* number of neurons */
            int time, /* integer time */
            float h, /* time step */
            float *g, /* lookup table */
            int t_max, /* size of lookup table */
            float w_in,float w_ex,int delay,
            float I, /* applied current */
            float sigma /* amplitude of Gaussian white noise */
           )
{
  int i,j,k,index;
  float sum,v_new[2],dv[2];
  int spike;
  for(i=0;i<n;i++)
  {
    sum=0; /* calculate interactions */
    for(j=0;j<net[i].connect.size;j++)
    {
      index=net[i].connect.index[j];
      /* iterating over the ring buffer of neuron index */
      for(k=0;k<net[index].spikes.len;k++)
      {
         spike=getrbuffer(&net[index].spikes,k)+delay;
         /* testing whether spike is active */
         if(time>=spike && time<spike+t_max)
         {
           /* yes */
           if(net[index].inhib)
             /* inhibitory coupling */
             sum-=w_in*g[time-spike]*(net[i].v[0]-V_in);
           else
             /* excitatory coupling */
             sum-=w_ex*g[time-spike]*(net[i].v[0]-V_ex);
         }
         else
           /* no, break if all spikes are not active anymore */
           if(spike+t_max<time)
```

```
            break;
        } /* of for(k=...) */
      } /* of for(j=...) */
      /* update of ML */
      updateml(dv,net[i].v,I);
      v_new[0]=net[i].v[0]+(h/c)*(dv[0]+sum);
      v_new[1]=net[i].v[1]+h*dv[1]+sqrt(h)*sigma*gasdev();
      /* critical threshold of zero reached for v? */
      if(net[i].v[0]<0 && v_new[0]>0)
      {
        /* yes, we have a spike event, add to my ring buffer */
        addrbuffer(&net[i].spikes,time);
        fprintf(file,"%g %d\n",time*h,i);
      }
      net[i].v[0]=v_new[0];
      net[i].v[1]=v_new[1];
    } /* of for(i=...) */
  } /* of 'update' */
```

The mean value of v_i over the n neurons as a diagnostic variable, $\bar{v} = \frac{1}{n}\sum_{i=1}^{n} v_i$, can be calculated using the `mean` function:

```
float vmean(const Neuron net[],int n)
{
  int i;
  float sum;
  sum=0;
  for(i=0;i<n;i++)
    sum+=net[i].v[0];
  return sum/n;
} /* of 'vmean' */
```

The main program calculating the dynamics of the network is as follows:

```
#include "list.h"
#include "neuron.h"
#define n 100 /* number of neurons */
#define I 0.1 /* input current */
#define sigma 0.0 /* amplitude of Gaussian white noise */
#define t_max 20.0 /* duration of spike (ms) */
#define h 0.01  /* time step  (ms) */
#define p_conn 1.0 /* fully coupled network */
#define p_inhib 0.0 /* no inhibitory coupling */
#define t_end 100.0 /* simulation time (ms) */
#define delay 0 /* no delay */

int main(int argc,char **argv)
```

```
{
  Neuron net[n];
  float *g,g1,w_in,w_ex;
  int t,nstep,ostep,tmax;
  FILE *file,*log;
  init(net,n,p_conn,p_inhib);
  w_in=0.1/((n-1)*p);
  w_ex=0.1/((n-1)*p);
  nstep=t_end/h;
  ostep=nstep/100;
  tmax=t_max/h;
  g=getg(tmax,h); /* creating lookup table for g */
  file=fopen("neuron.spike","wb");
  log=fopen("neuron.mean","w");
  for(t=0;t<nstep;t++) /* time loop */
  {
    if(t % ostep==0)
      /* write to output file every ostep time steps */
      fprintf(log,"%g %g\n",t*h,vmean(net,n));
    update(file,net,n,t,h,g,t_max,w_in,w_ex,delay,I,sigma);
  } /* of  time loop */
  fclose(file);
  fclose(log);
  return 0;
} /* of 'main' */
```

1.2.2 Parallel Version

The parallelization is based on `Pnet`. The data to be exchanged now have the type `MPI_INT`. In order to incorporate the ring buffer storing the spike events, a new datatype `Rnet` has to defined. `inhib` is not part of datatype `neuron`, because information about remote neurons is also needed.

```
typedef struct
{
  Pnet *pnet;   /* parallel network datatype */
  Rbuffer *rbuffer; /* event ring buffer */
  int *inhib;   /* inhibitory list */
} Rnet;
```

The datatype `Neuron` now contains only the last spike event of the neuron itself:

```
typedef struct
{
  float v[2];   /* state of neuron (ML) */
```

```
  int spike;    /* time of last spike */
} Neuron;
```

Both the initialization of the neuron and the parallel communication pattern are organized by the function `init`. The information about whether a remote neuron is inhibitory or not is distributed by a call to `pnet_exchg`. The output buffer contains boolean values.

```
#define NOFIRE -1
Neuron *init(Rnet *rn,
             int n, /* total number of neurons */
             float p_conn, /* probability of establishing
                              connection */
             float p_inhib /* probability of a inhibitory
                              neuron */
            ) /* returns allocated subarray of neurons */
{
  Neuron *net;
  int i,*buffer;
  int *inhib;
  rn->pnet=pnet_init(MPI_INT,n);
  /* setup of random network */
  pnet_random(rn->pnet,p_conn);
  /* allocate subarray of neurons and temp. inhibitory array*/
  net=newvec(Neuron,rn->pnet->lo,rn->pnet->hi);
  inhib=newvec(int,rn->pnet->lo,rn->pnet->hi);
  pnet_foreach(rn->pnet,i)
  {
    net[i].spike=NOFIRE;
    /* Set initial condition of neuron */
    net[i].v[0]=-0.02*0.01*drand48();
    net[i].v[1]=0.05+0.20*drand48();
    inhib[i]=(drand48()<p_inhib);
  }
  pnet_setup(rn->pnet);
  /* initialization of ring buffer  */
  rn->rbuffer=(Rbuffer *)malloc(sizeof(Rbuffer)*rn->pnet->insize);
  for(i=0;i<rn->pnet->insize;i++)
    initrbuffer(rn->rbuffer+i,RBUFF_LEN);
  /* allocating information about inhibitory neurons */
  rn->inhib=(int *)malloc(sizeof(int)*rn->pnet->insize);
  /* mapping inhibitory array to output buffer;
  buffer=(int *)rn->pnet->outbuffer;
  for(i=0;i<rn->pnet->outsize;i++)
    buffer[i]=inhib[rn->pnet->outindex[i]];
/* distributing inhibitory vector */
```

```
  pnet_exchg(rn->pnet);
  buffer=(int *)rn->pnet->inbuffer;
  for(i=0;i<rn->pnet->insize;i++)
    rn->inhib[i]=buffer[i];
  /* free temporary storage */
  freevec(inhib,rn->pnet->lo);
  return net;
} /* of 'init' */
```

In order to write out the timing and the corresponding index of the firing neuron in a sequential way, a datatype `Slist` has to be defined:

```
typedef struct
{
  int time,neuron;
} Spike;
typedef struct
{
  Spike *index;
  int len;  /* length of list */
} Slist;

/* Declaration of functions */

/* Initialize empty list */
extern void initspikelist(Slist *);
/* Add spike to list */
extern int  addspikelistitem(Slist *,Spike);
/* Empty list */
extern void emptyspikelist(Slist *);
```

Implementation of `Slist` is identical to `List`. If neuron fires, then the index together with the timing of the event is stored in the `Spike` structure and added to the spike list. The parallel update function can be written as:

```
void update(Slist *spikelist, /* Spike list */
            Rnet *rn,
            Neuron net[], /* array of neurons */
            int time, /* integer time */
            float h,  /* time step (ms) */
            float *g, /* lookup table */
            int t_max, /* size of lookup table */
            float w_in,float w_ex,int delay,
            float I, /* applied current */
            float sigma /* amplitude of Gaussian white noise */
           )
{
```

```
  int i,j,k,index;
  float sum,v_new[2],dv[2];
  int spike;
  Spike event;
  pnet_foreach(rn->pnet,i)
  {
    sum=0; /* calculate interactions */
    for(j=0;j<rn->pnet->connect[i].len;j++)
    {
      index=rn->pnet->connect[i].index[i];
      for(k=0;k<rn->rbuffer[index].len;k++)
      {
        spike=getrbuffer(rn->rbuffer+index,k)+delay;
        if(time>=spike && time<spike+t_max)
        {
          if(rn->inhib[index])
            sum-=w_in*g[time-spike]*(net[i].v[0]-V_in);
          else
            sum-=w_ex*g[time-spike]*(net[i].v[0]-V_ex);
        }
        else if(spike+t_max<time)
          break;

      }
    } /* of for(j=..) */
    updateml(dv,net[i].v,I);
    v_new[0]=net[i].v[0]+(h/c)*(dv[0]+sum);
    v_new[1]=net[i].v[1]+h*dv[1]+sqrt(h)*sigma*gasdev();
    /* critical threshold of zero reached for v? */
    if(net[i].v[0]<0 && v_new[0]>0)
    {
      /* yes, store time */
      net[i].spike=time;
      event.time=time;
      event.neuron=i;
      addspikelist(spikelist,event); /* add spike to list */
    }
    net[i].v[0]=v_new[0];
    net[i].v[1]=v_new[1];
  } /* of for(i=..) */
} /* of 'update' */
```

The exchange of spike timings is performed by a call of the `exchg` function:

```
void exch(Rnet *rn,Neuron net[])
{
```

```
  int i,*buffer;
  /* write time of last spike to output buffer */
  buffer=(int *)rn->pnet->outbuffer;
  for(i=0;i<rn->pnet->outsize;i++)
    buffer[i]=net[rn->pnet->outindex[i]].spike;
  pnet_exchg(rn->pnet); /* Communication */
  /* add times of last spike to the corresponding ring buffer */
  buffer=(int *)rn->pnet->inbuffer;
  for(i=0;i<rn->pnet->insize;i++)
    if(buffer[i]!=NOFIRE) /* spike occured */
       if(rn->rbuffer[i].len==0 ||
          rn->pnet->inbuffer[i]!=getrbuffer(rn->rbuffer+i,0))
          /* we have a new spike */
          addrbuffer(rn->rbuffer+i,buffer[i]);
} /* of 'exch' */
```

The exchange is only necessary after time t_{delay}. Therefore, communication is significantly reduced in comparison to diffusive coupled neurons.

Each task contains a list of all spike events occured stored locally in the `Slist` datatype. The serialized output of the spike events collected from all tasks is achieved by the `fwritespikes` function. All tasks initially send the number of events recorded via the collective `MPI_Gather` operation to task zero. Then the content of the spike list is sent via `MPI_Send` to task zero.

```
#define MSG_TIME 99 /* message tag used by send/recv */

void fwritespikes(FILE *file,Pnet *pnet,Slist *list)
{
  int len,i,*list_len;
  Spike *vec;
  MPI_Status status; /* needed by MPI_Recv */
  len=list->len;
  list_len=newvec(int,0,pnet->ntask-1);
  /* Gather number of spikes from all tasks */
  MPI_Gather(&len,1,MPI_INT,list_len,1,MPI_INT,
             0,MPI_COMM_WORLD);
  if(pnet->taskid==0)
  {
    /* write spike events of task 0 */
    fwrite(list->index,sizeof(Spike),len,file);
    /* collect spike events from all other tasks */
    for(i=1;i<pnet->ntask;i++)
      if(list_len[i]>0) /* spike occured in task i? */
      {
        /* yes, allocate temporal storage */
        vec=newvec(Spike,0,list_len[i]-1);
```

```
        /* receive spike list from task i */
        MPI_Recv(vec,sizeof(Spike)*list_len[i],
                 MPI_BYTE,i,MSG_TIME,
                 MPI_COMM_WORLD,&status);
        fwrite(vec,sizeof(Spike),list_len[i],file);
        free(vec);
      }
  }
  else if (len>0) /* spike occured in my task */
    /* send to task zero */
    MPI_Send(list->index,sizeof(Spike)*len,MPI_BYTE,
             0,MSG_TIME,MPI_COMM_WORLD);
  free(list_len);
} /* of 'fwritespikes' */
```

The function for calculating the mean value $\bar{v}$ in parallel uses the global reduction function `MPI_Reduce`. The reduction function `MPI_SUM` adds the values of all tasks. The function returns in task zero the global sum:

```
float vmean(Rnet *rn,
            const Neuron net[] /* subarray of neurons */
            ) /* returns mean value of v on task zero */
{
  int i;
  float sum,globalsum;
  sum=0;
  pnet_foreach(rn->pnet,i)
    sum+=net[i].v[0];
  /* global reduction of sum o globlasum on task zero
     using add operator */
  MPI_Reduce(&sum,&globalsum,1,MPI_FLOAT,MPI_SUM,
             0,MPI_COMM_WORLD);
  return globalsum/rn->pnet->n;
} /* of 'vmean' */
```

The main program of the parallel version is:

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h> /* MPI prototypes */
#include "list,h" /* list datatype */
#include "pnet.h"
#include "rnet.h"
#include "neuron.h"
int main(int argc,char **argv)
{
  Neuron *net;
```

```
float *g,g1,w_in,w_ex,v;
int t,nstep,ostep,tmax;
FILE *file,*log;
Rnet rnet;
Slist spikelist;
MPI_Init(&argc,&argv); /* initialize MPI */
net=init(&rnet,n,p_conn,p_inhib);
/* set random seeds differently for each task */
srand48(22892+38*rnet.pnet->taskid);
/* setting inhibitory and excitatory coupling strength */
w_in=0.1/((n-1)*p_conn);
w_ex=0.1/((n-1)*p_conn);
nstep=t_end/h;
ostep=nstep/100;
tmax=t_max/h;
g=getg(tmax,h);
if(rnet.pnet->taskid==0)
{
  /* opening output files on task 0 */
  file=fopen("neuron.spike","wb");
  log=fopen("neuron.mean","w");
}
initspikelist(&spikelist);
for(t=0;t<nstep;t++) /* time loop */
{
  if(t % ostep==0)
  {
    /* write mean value of v to output file every
       ostep time steps */
    v=vmean(&rnet,net);
    if(rnet.pnet->taskid==0)
      fprintf(log,"%g %g\n",t*h,v);
  }
  update(&spikelist,&rnet,net,t,h,g,t_max,
         w_in,w_ex,delay,I,sigma);
  if(delay==0 || t % (delay-1)==0)
  { /* exchange necessary every delay time steps */
    exch(&rnet,net);
    /* write out spike timings */
    fwritespikes(file,rnet.pnet,&spikelist);
    emptyspikelist(&spikelist);
  }
} /* of time loop */
if(rnet.pnet->taskid==0)
{
```

```
    fclose(file);
    fclose(log);
  }
  MPI_Finalize(); /* end MPI */
  return 0;
} /* of 'main' */
```

Sample output of the spiking times of a ML network is shown in Fig. 1.11.

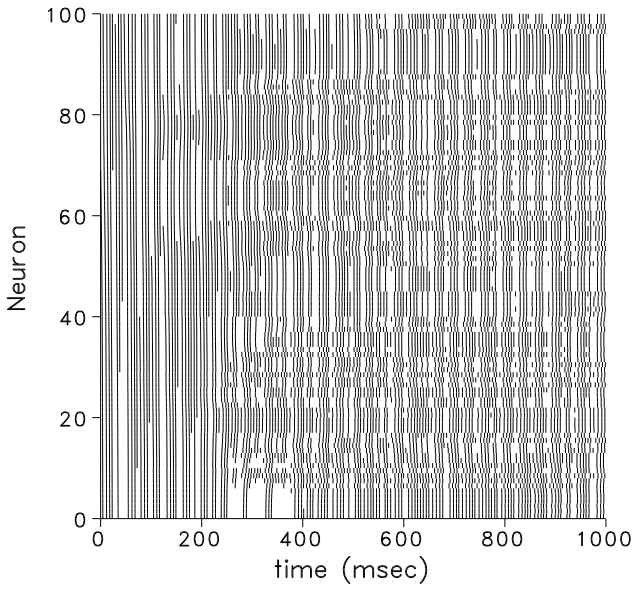

Fig. 1.11. Spike pattern for a network of 100 neurons derived from file `neuron.spike`

1.3 Connection Dependent Coupling Strengths and Delays

For the sake of simplicity, we have up to now only considered globally uniform values for the coupling strength $w_{\mathrm{in,ex}}$ and delays t_{delay}. In general, these are connection dependent values. This can be implemented by defining a new datatype for connections:

```
typedef struct
{
  int index; /* index of neuron */
  float w;   /* connection dependent weight */
  int delay; /* delay */
```

```
} Conn;

typedef struct
{
  Conn *conns; /* array of connections */
  int size;    /* number of connections */
} Connlist;
```

Then, the datatype `neuron` is defined as:

```
typedef struct
{
  float v[2];       /* state of neuron (ML) */
  Rbuffer spikes;   /* times of last spikes */
  int inhib;        /* inhibitory or not */
  Connlist connect; /* connection list */
} Neuron;
```

Using this data structure, it is possible to model spike time dependent plasticity, i.e. the weights are modified differently, dependent on the times of the pre- and postsynaptic spike arrival times t_i and t_j (Chapts. I and IV). The weight w_{ij} of a connection is increased or decreased by Δw_{ij} according to:

$$\Delta w_{ij} = \begin{cases} A_+ \exp(\Delta t/t_+) \text{ for } \Delta t > 0 \\ A_- \exp(\Delta t/t_-) \text{ for } \Delta t < 0 \end{cases}, \tag{1.14}$$

where $A_+ > 0$, $A_- < 0$ and $\Delta t = t_i - t_j$. The function Δw_{ij} is implemented

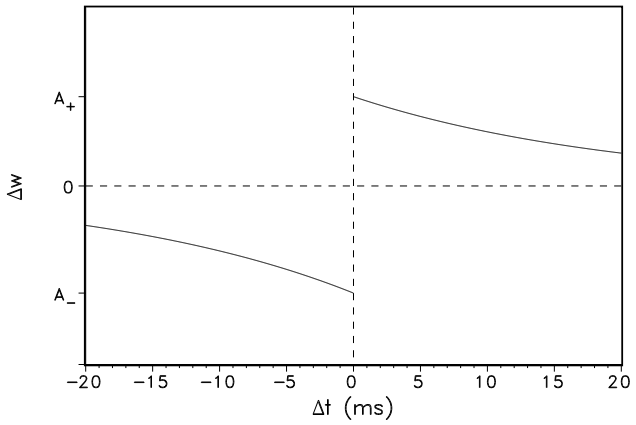

Fig. 1.12. Plasticity Δw as a function of difference between pre- and postsynaptic spike arrival Δt

in the following way:

```
#define A_plus 0.01
#define A_minus (-0.012)
#define t_plus 20.0
#define t_minus 20.0

float deltaw(float deltat)
{
  return (deltat>0) ? A_plus*exp(-deltat/t_plus)
                    : A_minus*exp(deltat/t_minus);
} /* of 'deltaw' */
```

Then the sequential update function with plasticity is:

```
void update(FILE *file, /* output file for spikes */
            Neuron net[], /* array of neurons */
            int n, /* number of neurons */
            int time, /* integer time */
            float h, /* time step */
            float *g, /* lookup table */
            int t_max, /* size of lookup table */
            float I, /* applied current */
            float sigma /* amplitude of Gaussian white noise */
           )
{
  int i,j,k,index,last;
  float sum,v_new[2],dv[2];
  int spike;
  for(i=0;i<n;i++)
  {
    sum=0; /* calculate interactions */
    for(j=0;j<net[i].connect.size;j++)
    {
      index=net[i].connect.conns[j].index;
      /* iterating over the ring buffer of neuron index */
      for(k=0;k<net[index].spikes.len;k++)
      {
        spike=getrbuffer(&net[index].spikes,k)+
              net[i].connect.conns[j].delay;
        /* testing whether spike is active */
        if(time>=spike && time<spike+t_max)
        {
          /* yes */
          if(net[index].inhib)
            /* inhibitory coupling */
            sum-=net[i].connect.conns[j].w*
                 g[time-spike]*(net[i].v[0]-V_in);
```

```
      else
        /* excitatory coupling */
        sum-=net[i].connect.conns[j].w*
             g[time-spike]*(net[i].v[0]-V_ex);
      /* plasticity */
      if(time==spike)
      {
        /* previous spike occured? */
        if(net[i].spikes.len>0)
        {
          /* yes, change weight of connection, delta t<0 */
          last=getqueue(net[i].spikes,0);
          net[i].connect.conns[j].w+=
             deltaw(-(spike-last)*h);
        }
      }
      else
        /* no, break if all spikes are not active anymore */
        if(spike+t_max<time)
          break;
  } /* of for(k=...) */
} /* of for(j=...) */
/* update of ML */
updateml(dv,net[i].v,I);
v_new[0]=net[i].v[0]+(h/c)*(dv[0]+sum);
v_new[1]=net[i].v[1]+h*dv[1]+sqrt(h)*sigma*gasdev();
/* critical threshold of zero reached for v? */
if(net[i].v[0]<0 && v_new[0]>0)
{
  /* yes, we have a spike event, add to my ring buffer */
  addrbuffer(&net[i].spikes,time);
  fprintf(file,"%g %d\n",time*h,i);
  /* plasticity */
  for(j=0;j<net[i].connect.len;j++)
  {
    index=net[i].connect.conns[j].index;
    for(k=0;k<net[index].spikes.len;k++)
    {
      spike=getqueue(net[index].spikes,k)+
            net[i].connect.conns[j].delay;
      if(spike<time)
      {
        /*change weight of connection, delta t>0 */
        net[i].connect.conns[j].w+=deltaw((time-spike)*h);
        break;
```

```
          }
        } /* of for(k=...) */
      } /* of for(j=...) */
    }
    net[i].v[0]=v_new[0];
    net[i].v[1]=v_new[1];
  } /* of for(i=...) */
} /*  of 'update' */
```

The initialization of the network has to include setting up the connection-dependent weights and delays. The parallel version of the code can be implemented analogously. The parallel exchange performed by `pnet_exch` is only necessary every $t_{\min} \times \Delta t$ time steps, where $t_{\min}$ is defined as

$$t_{\min} = \min_{i,j=1\ldots n} t_{\text{delay},i,j}. \tag{1.15}$$

References

[1] R. Sedgewick: *Algorithms in C, Part 5: Graph algorithms*, 3rd edn (Addison Wesley Reading MA 2001)

[2] C Morris, H. Lecar: Biophys. J., **35**, 193 (1981)

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: *Introduction to algorithms*, 2nd edn (MIT Press Cambridge MA 2001)

[4] B W. Kernighan, D. Ritchie: *The C programming language* (Prentice Hall 1988)

[5] W. Gropp, E. Lusk, A. Skjellum: *Using MPI: Portable parallel programming with the message-passing interface*, 2nd edn (MIT Press Cambridge MA 1999)

[6] A. Morrison, C. Mehring, T. Geisel, A. Aertsen, M. Diesmann: Neural Computation, **17**, 1776 (2005)

[7] A. Tam, C. Wang: Efficient scheduling of complete exchange on clusters. In: *13th international conference in parallel and distributed computing systems* (PCDS Las Vegas 2000)

[8] G. M. Amdahl: Validity of the single-processor approach to achieving large scale computing capabilities. In: *AFIPS Conference Proceedings*, vol. 30, (AFIPS Press Reston VA 1967) pp. 483–485.

[9] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: *Numerical recipes in C: The art of scientific computing*, 2nd edn (Cambridge University Press Cambridge MA 1992)

Part V

Applications

Index