High Performance Ring Networks

ARACHNE

This manual describes how to set up and run MPI+OMP version of ring networks ARACHNE on a computers cluster. The general scenario of the simulation is as follows. A user works on a host computer using MATLAB installed under Windows. MATLAB program generates input MAT-file and uploads it to remote HPC cluster operated under Linux. Then the cluster part starts a simulation without the communications with the host computer. The user’s computer connects to the remote process from time to time to a) know what current number of iteration is and b) download intermediate results of the simulation (the results are shown in MATLAB Plots and save to the output file). When simulation completes, MATLAB host program downloads the output MAT-file and visualises the final results on the host computer.

Contents

[Preparing for the first launch 2](#_Toc397896450)

[Local machine side (under Windows) 2](#_Toc397896451)

[HPC cluster side (under Linux) 2](#_Toc397896452)

[Running the simulation 3](#_Toc397896453)

[Single experiment 3](#_Toc397896454)

[Scalability test 3](#_Toc397896455)

[Non-remote simulation modes 4](#_Toc397896456)

[Changing ARACHNE parameters 5](#_Toc397896457)

[The file “ModelParams.m” 5](#_Toc397896458)

[Changing parameters and formula of Hebbian correction 7](#_Toc397896459)

[Changing deterministic external drives to E- and I-cells 7](#_Toc397896460)

[Synaptic connections localization models 7](#_Toc397896461)

[The file “HpcParams.m” 9](#_Toc397896462)

[Changing system of equations 15](#_Toc397896463)

[Choosing and initialization of random number generators 16](#_Toc397896464)

[Changing parameters in the case of simulation continuation 17](#_Toc397896465)

[Model restrictions, memory requirements and performance 18](#_Toc397896466)

[Bit synaptic conductance matrices 18](#_Toc397896467)

[Memory allocated on a cluster node 18](#_Toc397896468)

[Limitations of the number of neurons caused by memory requirements 20](#_Toc397896469)

[Performance of ARACHNE 20](#_Toc397896470)

[Scalability curves for different types of matrices 21](#_Toc397896471)

[Structure of MATLAB host program 22](#_Toc397896472)

[Structure of HPC kernel 23](#_Toc397896473)

[Source code 23](#_Toc397896474)

[I/O directories 25](#_Toc397896475)

[Passing new parameters from host (MATLAB) to workers (C++) 26](#_Toc397896476)

[Passing a new scalar to an equation 26](#_Toc397896477)

[Passing a new vector to an element-wise operation 26](#_Toc397896478)

[The case of scattering of the vector 26](#_Toc397896479)

[The case of broadcasting of the vector 27](#_Toc397896480)

# Preparing for the first launch

## Local machine side (under Windows)

Download two directories ***host*** and ***worker*** from GITHUB repository

<https://github.com/LeonidSavtchenko/Arachne>

to a local computer:

1. Copy the directory “<root>/host at any place of your local computer with Windows operating system and preinstalled MATLAB.
2. Copy the directory/worker on your cluster. ( See details “How to run a simulation on a remote cluster”)

To test the model ARACHNE, the cluster, located at 144.82.46.83, is already prepared. The following directory contains preinstalled software:

/home/reviewer/worker

Password: reviewer1

In this case to start a simulation with the configuration:

1. Go to the directory … “<root>/host at your local computer
2. Launch the following Matlab script: “<root>/host/START\_GammaSimulator.m”.
3. When GUI appears, and then check the parameters of the model. At this stage you can change the settings.
4. After that you can click “OK” -- the simulation will be conducted on the remote cluster at 144.82.46.83.
5. When the simulation is completed, there will be plot with the simulation results

If you want to reinstall the application on this cluster or to install it on other Linux clusters you need to do the following.

## Preparing HPC cluster (under Linux)

1. Before installation the software, one need to be sure that “mpic++” compiler is present in the system path.
2. Download the folder “worker” with all its content from the GitHub repository. This folder must be saved to the place shared between cluster nodes. For example, it can be saved in the directory “/home/<username>”.
3. Compile the application running script “<root>/worker/build/lin\_release.sh.” If the compilation is successful, the file “gs.exe” must appear in the parental directory.

## Preparing your local computers (under Windows)

1. Make sure that Matlab is installed on the local computer.
2. Download the folder “host” with its contents from GitHub repository to the local machine.
3. Open file “<root>\host\Code\scripts\win-lin\params.bat” and adjust the following 4 variables: : “HEADNODEIP” – IP address of the cluster, “LOGIN” – Login name of your account of the cluster, “PASSWORD” – password of your account, “HEADNODEWORKERDIR” – the address of the location of “***gs.exe***” file on the cluster, to be consistent with your cluster.
4. Open file “<root>\host\Code\BasicParams\BasicParams.m”, set remoteHPC = true and initialize “availableNodes” cell array with your cluster node names.
5. Launch the following Matlab script: “<root>\host\START\_GammaSimulator.m”.
6. When GUI appears, go to “HPC” panel and adjust the following three elements:

* checkbox “fakeMPI” -- unchecked,
* editbox “nt” = the number of processor cores per one cluster node,
* checkbox “backgroundMode” -- checked.

1. After that you can click “OK” -- the simulation will be conducted on the remote cluster.

# How to run the simulation on your local computer without cluster.

1. Make sure that you have Matlab installed on the machine.
2. Make sure that you have Visual Studio Community installed on the machine. This free IDE can be downloaded here:

<https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>

The “Visual C++” option must be checked during installation.

1. Download the folders “host” and “worker” keeping the structure and all of its content from GitHub repository to your local machine.
2. Open file “<root>\worker\build\vars.bat” in any text editor and adjust the following three paths to be consistent with your machine: “VSDIR”, “MLDIR”, “GSDIR”. VSDIR is a director with Visual Studio, MLDIR is a director with Matlab, GSDIR is a directory with gs.exe
3. Run script “win\_fakeMPI\_release.bat” located in this directory. The file “gs\_fakeMPI.exe” should appear in the parental directory after that.
4. Open file “<root>\host\Code\scripts\win-win\params.bat” and adjust the following 2 paths: “MATLABHOSTDIR”, “WORKERDIR”.
5. Open file “<root>\host\Code\BasicParams\BasicParams.m” and set remoteHPC = false.
6. Launch the following Matlab script: “<root>\host\START\_GammaSimulator.m”.
7. When GUI appears, go to “HPC” panel and adjust the following three elements:

* checkbox “fakeMPI” -- checked,
* editbox “nt” = the number of cores in your processor,
* checkbox “backgroundMode” -- unchecked.

1. After that you can click “OK” -- the simulation will be conducted on your local machine instead of the cluster.

# Running the simulation

## Single experiment

1. Wake up all nodes of the cluster. (Alternatively, you can run simulation on the master node only.)
2. Open file “host\Params\HpcParams.m” and specify *remoteHPC = true*.

Specify number of processes “np” you are going to run on the cluster. (The number should not exceed the total number of processors on available cluster nodes.) Specify number of threads per process “nt”. (The number should not exceed the number of cores of a processor.)

1. Open file “host\START\_GammaSimulator.m” in MATLAB and press F5.

If everything is OK, you will see messages in MATLAB console reporting about:

1. Uploading of input MAT-file to the head node of the cluster.
2. Progress of simulation.
3. Downloading of output MAT-file from the head node.

Then windows with graphical representation of the results must appear.

If you need to stop the simulation, click on Command Window in MATLAB and press Ctrl + Break. This stops the client (local computer) part of simulation, but not the server performance. In order to stop server operations, one needs to run the first or the second script from the following:

1. “SCRIPT\_TerminateBackgroundProcess.m”.
2. “SCRIPT\_KillBackgroundProcess.m”.

## Scalability test

1. Open file “host\Params\HpcParams.m” and specify *scalTest = true*.

Specify maximum number of processes “np” and maximum number of threads per process “nt” you are going to use in the test.

1. Specify the minimum number of processes “minNP” taking into account that a very large number of neurons and too small number of processes may cause memory deficit (see the paragraph below). On the other hand, make sure that the number of neurons is large enough to show the real scalability.

Here are some examples:

1. minNP = 1; np = 9; nt = 4; num\_e = 200; num\_i = 100; t\_final = 500; radius\_e = 250; radius\_i = 200; v = 0.1; scmType\_XY = KrnDense; distMatPVH = false;

BAD — not enough neurons to show real scalability;

1. minNP = 1; np = 9; nt = 4; num\_e = 8600; num\_i = 4300; t\_final = 0.2; radius\_e = 250; radius\_i = 200; v = 0.1; scmType\_XY = KrnDense; distMatPVH = false;

BAD — memory deficit happens for number of processes less than 5 (the amount of required physical memory per cluster node exceeds 1700 MB);

1. minNP = 8; np = 9; nt = 4; num\_e = 8600; num\_i = 4300; t\_final = 0.2; radius\_e = 250; radius\_i = 200; v = 0.1; scmType\_XY = KrnDense; distMatPVH = false;

GOOD.

1. Open file “host\START\_GammaSimulator.m” in MATLAB and press F5.

If everything is OK, you will see messages in MATLAB console reporting that simulation is run for number of processes equal *minNP, minNP + 1, …, np* and number of threads per process equal *1, 2, …, nt* in sequence.

At the end the window must appear showing set of scalability curves. Each curve y(x) corresponds to specific number of threads and is defined by “x” being number of processes and “y” being inverse time of execution.

# Non-remote simulation modes

Besides the remote simulation mode described above, the simulator can be run in the following two modes:

1. Running from MATLAB on local machine under Windows on the same machine.
2. Running from MATLAB on head node of Linux cluster on the same cluster.

These two modes were developed mostly for debugging and testing purposes. In order to use the first mode, you need to adjust variables defined in the script “host\scripts\win-win\params.bat.” Usage of the second mode requires adjustment of variables defined in “host\scripts\lin-lin\params.sh.” The parameter “remoteHPC” defined in “host\Params\HpcParams.m” should be specified as “false” in both cases.

The first mode needs compilation of HPC kernel under Windows. The compilation requires the following software installed:

1. Microsoft Compute Cluster Pack SDK available for free by the following reference:

<http://www.microsoft.com/en-us/download/details.aspx?id=239>

1. Microsoft Visual Studio 2012 or other C++ compiler. Notice that compilation with other C++ compiler may require changes in the code of HPC kernel because each compiler has its own specific requirements to the code.
2. MATLAB R2013a. Usage of some other version may require changes in code of HPC kernel.

MATLAB is necessary because it provides compile-time libraries and run-time libraries used for MAT-file read/write staff. The directory with MATLAB libraries must be added to the system path variable. You can access the variable as following:

1. Go to Windows Explorer -> My Computer -> Properties -> Advanced -> Environment Variables.
2. Select the variable with name “Path” and click “Edit” button.
3. Add “C:\Program Files\MATLAB\R2013a\bin\win64” to the end of the string. (Your addition to the string will be other if you installed MATLAB to some non-default location.)

Alternatively, you can copy stuff of the directory “C:\Program Files\MATLAB\R2013a\bin\win64” to the directory containing ARACHNE executable “gs.exe.”

Compilation of HPC kernel under Windows is performed with one of the following scripts located in the directory “worker”:

1. build\_Windows\_DEBUG.bat.
2. build\_Windows\_RELEASE.bat.

# Changing ARACHNE parameters

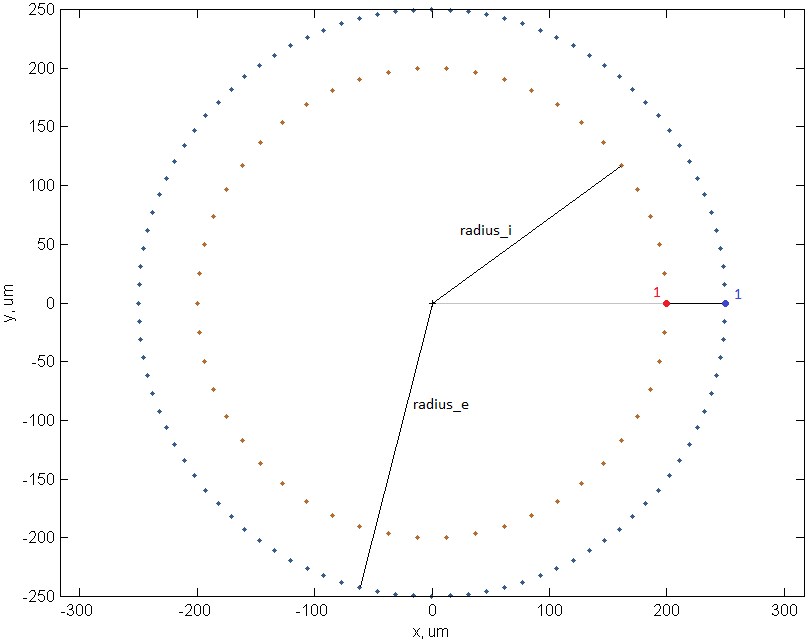
## The file “ModelParams.m”

This file corresponds to the file “params.m” of the original software. A group of new parameters was added to support new features of ring networks ARACHNE and some model parameters were moved to other place or deleted.

The new model parameters are as follows:

1. **radius\_e, radius\_i**

These two variables specify the radii of ring networks in micrometres. Variable “radius\_e” determines radius of e-network and variable “radius\_i” specifies radius of i-network. The numbering of neurons in ring networks guarantees that the distance between the first neuron of e-network and the first neuron of i-network is minimal (see picture 1).



Picture 1. *radius\_e = 250*, *radius\_i = 200*

1. **v**

The parameter specifies the rate of signal propagation in micrometres per millisecond.

1. **sclModel**

The parameter determines which synaptic connections localization model will be used.

BSS — bell-shaped strength and uniform density of connections.

BSD — uniform strength and bell-shaped density of connections.

Notice that parameters “g\_hat\_XY” with “XY” being “ee”, “ei”, “ie” and “ii” have an influence on the model only when *sclModel = BSD* or *scmType\_XY = AllEqual*.

For more information about synaptic connections localization models see [the next paragraph](#_Synaptic_connections_localization).

1. **w\_XY\_max** with XY being ee, ei, ie and ii

These four parameters specify the maximum heights of corresponding bells.

1. **sigma\_XY** with XY being ee, ei, ie and ii

These four parameters specify the standard deviations of corresponding normal distributions.

1. **useHC**

The parameter specifies whether to correct synaptic conductance matrices according to Hebbian theory (spike-timing-dependant plasticity).

true — to do Hebbian correction for all synaptic conductance matrices.

false — not to do Hebbian correction.

By default the formula defining how the synaptic strength is modified due to the position of Action potential is the following:

.

If you need to change this formula, see the paragraph [below](#_Changing_parameters_of) for instructions.

*Parameters 7-8 are used only if useHC = true*

1. **hcPeriod**

The parameter specifies how often the Hebbian correction is done.

The correction is done once per “hcPeriod” iterations; each time only impulses appeared on last “hcPeriod” iterations are analysed.

1. **hcParams**

The variable contains all parameters of Hebbian correction. It is a matrix with four rows and arbitrary number of columns. Each row of matrix “hcParams” contains parameters of Hebbian correction for one synaptic conductance matrix. By default there are eight parameters for each matrix. If you need to change the number of Hebbian correction parameters, see the paragraph [below](#_Changing_parameters_of) for instructions.

Four densities of synaptic connection “p\_XY” with “XY” being “ee”, “ei”, “ie” and “ii” were removed from the model because they have no influence on synaptic conductance matrices of ring networks.

One more change is that 2 parameters (functions) were moved to other place. These functions are:

1. I\_i(t) — deterministic external drive to I-cells.
2. I\_e(t) — deterministic external drive to E-cells.

The MPI+OMP version of the simulator does not define external drives in MATLAB anymore. They migrated to C++ code of HPC kernel for performance. The drive arrays returned by the functions became distributed vectors. (In other words, each node of the cluster keeps only some part of each vector.) The new home of the functions is the file “worker/ExternalDrives.cpp.”

By default, the drives are constants defined as follows: I\_e(t) = 0.5, I\_i(t) = 0.0 for each element of vector. If you need to change the default drives, see the paragraph [below](#_Changing_deterministic_external) for instructions.

Calculation of variables “m\_steps”, “delta\_ee”, “delta\_ei”, “delta\_ie”, “delta\_ii”, “max\_delay\_e” and “max\_delay\_i” was moved to this file from the file “gamma\_simulator.m” of prototype. Notice that when set of parameters “num\_e”, “num\_i”, “radius\_e”, “radius\_i” and “v” is unbalanced, i.e. at least one of variables “delta\_XY” is equal to zero, results of computation can be incorrect.

## Changing parameters and formula of Hebbian correction

1. Open the file “host\Params\ModelParams.m” and change the matrix “hcParams”.
2. Change or delete the validation of Hebbian correction parameters located after definition of this matrix.
3. Log in to the tuxmaster node and open the following file:

“/home/reviewer/gs/worker/DistMatrix.h”.

1. Update the code for the function “AtomicCorrection” as you need. Notice that in C++ the numbering of array elements begins from zero.
2. Compile HPC kernel running script “build\_Linux\_RELEASE.sh” living in the same directory.

If compilation is successful, the MATLAB host program being run for the next time will use the new just compiled HPC kernel.

## Changing deterministic external drives to E- and I-cells

1. Log in to the tuxmaster node and open the following file:

“/home/reviewer/gs/worker/ExternalDrives.cpp”.

1. Update the code for the functions “ComputeExternalDrive\_e” and “ComputeExternalDrive\_i” as you need.

You can follow the example provided in the file “ExternalDrives.cpp” for the case of “I\_e” being

I\_e(t, idx) = sin(PI **·** idx / (num\_e – 1)) **·** exp(–0.5 **·** t)

with “idx” being end-to-end index of the distributed vector: idx = 0, 1, …, num\_e – 1.

1. Compile HPC kernel running script “build\_Linux\_RELEASE.sh” living in the same directory.

If compilation is successful, the MATLAB host program being run for the next time will use the new just compiled HPC kernel.

## Synaptic connections localization models

There are two models of synaptic connections localization. In both model the normal distribution of one synaptic connection parameter is used:

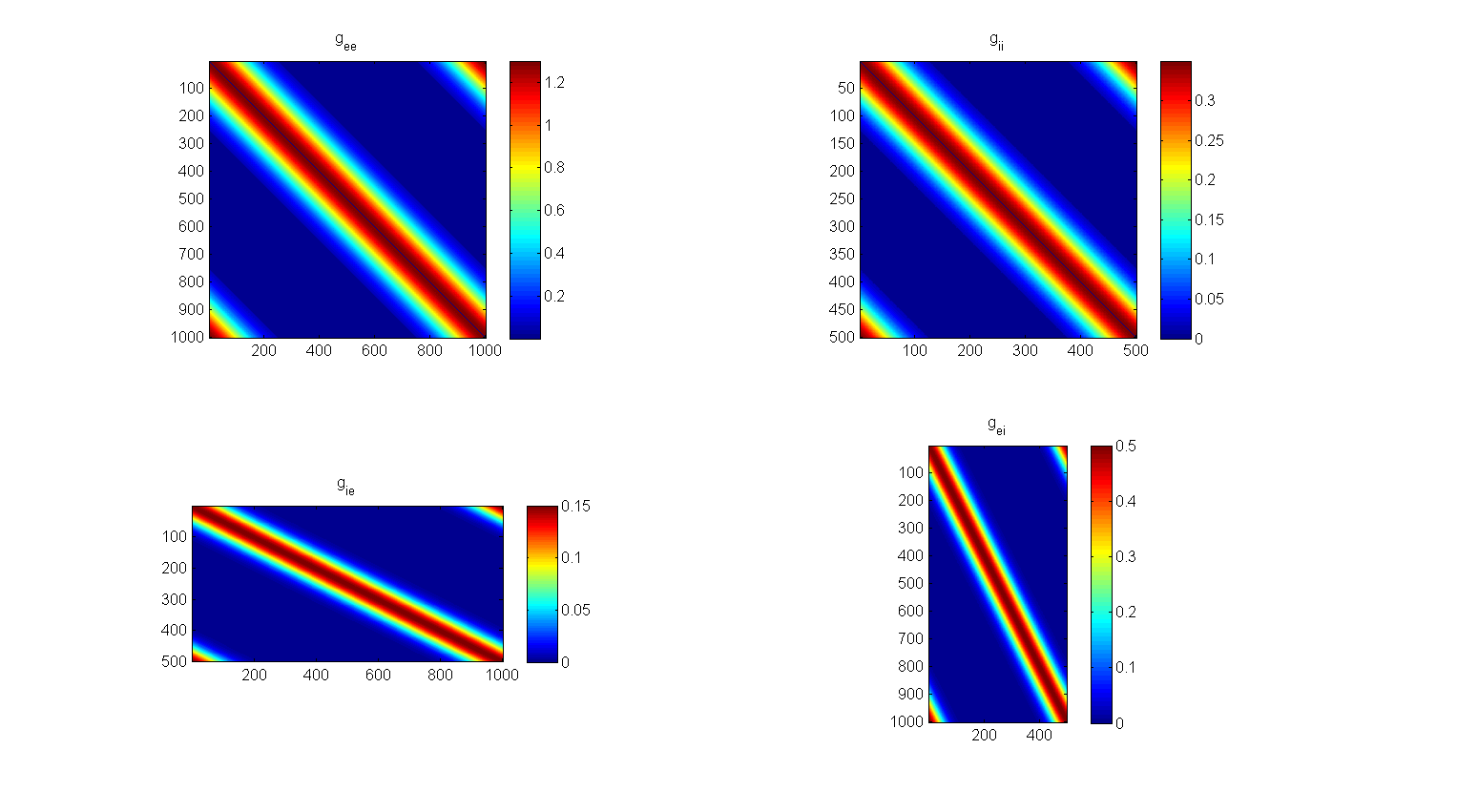
.

The first model consists in bell-shaped strength and uniform density of connections. Synaptic conductance matrices of this model are presented at the picture 2 with following set of parameters:

num\_e = 1000; num\_i = 500;

w\_ee\_max = 1.3; w\_ii\_max = 0.35; w\_ei\_max = 0.5; w\_ie\_max = 0.15;

sigma\_ee = 100; sigma\_ei = 75; sigma\_ie = 75; sigma\_ii = 50.



Picture 2

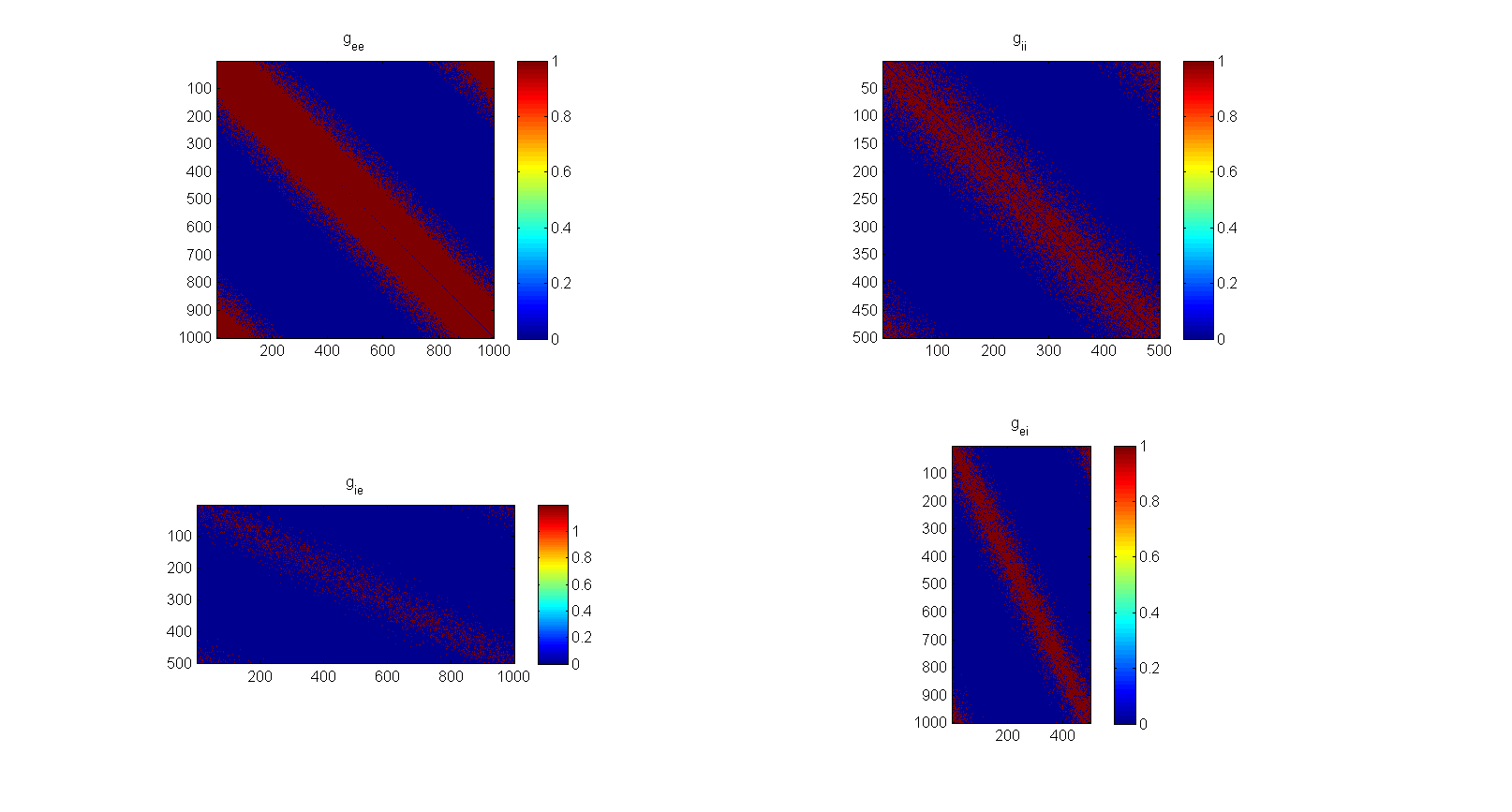
The second model consists in uniform strength and bell-shaped density of connections. If in this model the parameter “w\_XY\_max” is more than unit, the density of connections between neurons becomes saturated (compare matrices “g\_ee” and “g\_ie” at the picture 3). Synaptic conductance matrices of this model are presented at the picture 3 with following set of parameters:

num\_e = 1000; num\_i = 500;

g\_hat\_ie = 1.2; g\_hat\_ei = 1.0; g\_hat\_ii = 1.0; g\_hat\_ee = 1.0;

w\_ee\_max = 2; w\_ii\_max = 0.8; w\_ei\_max = 0.9; w\_ie\_max = 0.3;

sigma\_ee = 100; sigma\_ei = 75; sigma\_ie = 75; sigma\_ii = 50.



Picture 3

In this model of localization of synaptic connections the value w\_XY\_max >= 1 gives saturated density of connections.

In both models the elements on principal diagonals of intranetwork matrices (“g\_ee” and “g\_ii”) are always equal to zero to avoid influence of neuron on itself.

## The file “HpcParams.m”

This file contains the new parameters added specially to adjust High Performance Computing settings of ARACHNE.

The parameters are as follows:

1. **remoteHPC**

The variable specifies whether to use the remote HPC server.

true — to call HPC kernel on remote machine/cluster.

false — to call HPC kernel on this machine/cluster (see [the paragraph](#_Non-remote_simulation_modes) above).

1. **np**

The parameter specifies the number of parallel processes to launch.

When you’re changing it and going to run simulation on the cluster, make sure that “np” does not exceed the number of available cluster nodes. The number of available nodes is specified in the files “hostfile\_BusyMaster” and “hostfile\_IdleMaster” living in the directory “/home/reviewer/gs/worker/hostfiles”. The first file corresponds to the case when the master node works the same as a slave; the second file is for the case when the master node is not loaded. In the case of scalability test (see “scalTest” parameter below) “np” specifies the maximum number of processes in the test.

1. **nt**

The parameter specifies the number of threads per process.

While one process is run on one processor, one thread is run on one core of the processor. The best performance is achieved when the number of processes equals number of cluster nodes (i.e. the total number of processors) and the number of threads per process equals number of processor cores. In the case of scalability test “nt” specifies the maximum number of threads per processes in the test.

1. **scmType\_XY** with XY being ee, ei, ie, and ii

These 4 parameters specify the types of synaptic conductance matrices. The type of a matrix determines how and where the matrix will be created and populated, and what algorithm will be used for computation of the current on actual iteration.

Types **AllZeros** and **AllEqual**:

**AllZeros** — all matrix elements are zeroes;

**AllEqual** — all matrix elements are equal.

The matrix is not generated explicitly because it has primitive structure. The main advantages of these types are:

a) matrix can have giant size not requiring a lot of memory;

b) calculation of the current on actual iteration has excellent performance.

But it’s clear that these models of synaptic conductance matrix are too simple and make sense only in combination with other types.

Notice that these two types of matrices can be used only when *useHC = false* (see [the paragraph](file:///C:\Users\lsavtchenko\AppData\Roaming\Skype\My%20Skype%20Received%20Files\ModelParams.m#_The_file_) above).

Types **HstDense** and **HstSparse**:

**HstDense** — the matrix is generated in a dense form in MATLAB;

**HstSparse** — the matrix is generated in a sparse form in MATLAB.

The matrix is generated in MATLAB host program, saved to the input MAT-file with other data, uploaded to the head node of the cluster and scattered among all nodes. The difference between dense and sparse storage scheme is that only nonzero elements positions and values are stored in the second case. It provides memory economy in the case if the number of nonzero elements is much less than the number of zeroes.

The main advantage of these two types is that the matrices can be populated in MATLAB in variety of ways. There is no restriction on number of different values in a matrix. There is no need to recompile HPC kernel if algorithm of matrix population is changed.

The disadvantages are as follows:

a) amount of physical memory on local machine and head node of cluster puts limitation on matrix size;

b) uploading big matrices to the head node may take a lot of time;

c) scattering matrices from the head node to all nodes takes time as well.

Notice that type “HstSparse” can be used only when *sclModel = BSD* and *useHC = false* (see [the paragraph](file:///C:\Users\lsavtchenko\AppData\Roaming\Skype\My%20Skype%20Received%20Files\ModelParams.m#_The_file_) above).

Types **KrnDense** and **KrnSparse**:

**KrnDense** — the matrix is generated in distributed dense form in HPC kernel;

**KrnSparse** — the matrix is generated in distributed sparse form in HPC kernel.

The matrix is generated in HPC kernel before 1st iteration already in distributed form.  
These types are counterparts of types HstDense and HstSparse that provide the advantage that there is no need to upload and scatter big matrices. Since matrices are generated already in distributed form, the total amount of memory required by a matrix is divided by number of cluster nodes. As a result, bigger matrices can be generated in comparison with types HstDense and HstSparse. The disadvantage is that there is no such flexible way to populate synaptic conductance matrices. If any changes in the algorithm, HPC kernel should be recompiled.

Notice that type “KrnSparse” can be used only when *sclModel = BSD* and *useHC = false* (see [the paragraph](file:///C:\Users\lsavtchenko\AppData\Roaming\Skype\My%20Skype%20Received%20Files\ModelParams.m#_The_file_) above).

Type **KrnOneBit**:

An element of the matrix is stored in one bit.

The matrix has dense structure and is generated in HPC kernel before 1st iteration already in distributed form.

The advantage of this mode compared with “KrnDense” type is that the matrix stored in bit-packed format requires significantly less memory than the matrix stored in the floating-point format. The matrix of type “KrnOneBit” can be populated with only two different values: zero value and some other value.

The feature of type “KrnOneBit” in comparison with other ones is that it requires rounding of number of rows in the matrix: the number of rows has to be evenly divisible by 64. If this is not the case, the rounding is performed automatically to the nearest integer that fits the requirement. As a result, the number of neurons used in simulation becomes less.

Notice that type “KrnOneBit” can be used only when *sclModel = BSD* and *useHC = false* (see [the paragraph](file:///C:\Users\lsavtchenko\AppData\Roaming\Skype\My%20Skype%20Received%20Files\ModelParams.m#_The_file_) above).

Type **KrnInPlace**:

Each element is generated each time it’s used.

The matrix is re-generated on each iteration once again. The random number generator that produces the sequence of elements of the matrix is reseed with the same seed before each computation of the matrix-to-vector product. This type of synaptic conductance matrix provides vanishing limitations of the model size, but the performance of this type is the worst among all other types. The reason is that the call of random number generator for each element takes considerable time in total.

Notice that type “KrnInPlace” can be used only when *useHC = false* (see [the paragraph](file:///C:\Users\lsavtchenko\AppData\Roaming\Skype\My%20Skype%20Received%20Files\ModelParams.m#_The_file_) above).

1. **scalTest**

On/off scalability test.

true — to run HPC kernel for number of processes equal minNP, minNP + 1, …, np and number of threads per process equal 1, 2, …, nt. Plot set of scalability curves at the end of the test.

false — to run HPC kernel for number of processes equal “np” and number of threads per process equal “nt” only.

*Parameter 6 is used only if scalTest = true*

1. **minNP**

The parameter specifies the minimum number of processes (cluster nodes) used in scalability test. (The maximum number equals np.) Usage of this parameter is worth when

1. memory deficit does not allow running scalability test for small number of loaded cluster nodes;
2. testing for small number of processes is expected to be too long or does not provide an interest.
3. **showSCM**

The parameter specifies whether to show synaptic conductance matrices.

true — to show plots of synaptic conductance matrices at the end of simulation.

false — not to show matrices.

If this option is enabled, the program can require more memory (see the paragraph [below](#_Memory_requirements)).

Notice that this parameter can be equal “true” only when *useHC = true* (see [the paragraph](file:///C:\Users\lsavtchenko\AppData\Roaming\Skype\My%20Skype%20Received%20Files\ModelParams.m#_The_file_) above).

1. **useSPA**

The parameter specifies whether to use Single Precision Arithmetics instead of double precision one.

true — to use float data type in all computations. (4-byte floating-point numbers provide approx. 7 decimal digits accuracy.)

false — to use double data type. (8-byte floating-point numbers provide approx. 16 decimal digits accuracy.)

In general, usage of floats instead of doubles leads to the better performance and worse accuracy.

But in the case of ARACHNE the worse accuracy is not relevant because the algorithm of simulation is so that the noise is introduced into the data on each iteration explicitly. Therefore, the difference between single-precision results and double-precision ones can be considered as a part of the intrinsic noise.

*Parameter 9 is used only if useSPA = false*

1. **use32BitRng**

The parameter specifies whether to use coarse-grained 32-bit random number generator or fine-grained 64-bit one.

true — to use coarse-grained 32-bit random number generator.

false — to use fine-grained 64-bit random number generator.

The 32-bit generator (std::mt19937) provides higher performance and worse quality of random numbers, while the 64-bit generator (std::mt19937\_64) provides lower performance and better quality of the numbers. Usage of the 64-bit generator makes sense only with double precision arithmetics (*useSPA = false*). See also [the paragraph](#_Choosing_and_initialization) below.

1. **distMatPVH**

The parameter specifies in which mode matrices of presynaptic voltage history are held.

true — matrices are held in distributed mode (each node of cluster keeps only part of each matrix).

false — matrices are held in local mode (each node of cluster keeps each matrix in whole).

If this option is enabled, amount of required physical memory per cluster node decreases, but the time of simulation session considerably increases due to communication between processes.

Use *distMatPVH = true* only if there isn’t enough memory for local mode of presynaptic voltage history matrices. In this case the number of processes has to be as little as it is possible.

1. **stabAnalysis**

On/off stabilization analysis mode.

true — to compute “Frequency\_e”, “Frequency\_i”, “syncoef\_e”, “syncoef\_i” on each iteration and to show plots at the end of simulations.

false — to compute “Frequency\_e”, “Frequency\_i”, “syncoef\_e”, “syncoef\_i” just once at the end of simulations.

Notice that computation of the parameters on each iteration affects performance.

1. **reportPeriodIter**

The parameter specifies how often HPC kernel should report its progress (i.e. the number of current iteration and the duration of an iteration). The report is done once per “reportPeriodIter” iterations.

1. **saveIntermMat**

The parameter specifies whether to save intermediate data file “intermediate.mat” that makes it possible to continue simulation from the same point after completion or termination.

The file is saved when:

1. specified t\_final is reached;
2. termination is requested by user;
3. *saveBackupMats = true* and current iteration number is evenly divisible by backupPeriodIter;
4. simulation is working in background mode and user runs the script SCRIPT\_GetSnapshot.m (see [the paragraph](#_Structure_of_Matlab) below).

Notice that for avoidance of unpractical usage of physical memory this parameter can be equal “true” only when *distMatPVH = false*.

1. **saveBackupMats**

The parameter specifies whether backup files should be saved periodically. Saving backup files makes it possible to recover simulation progress in the case of abnormal termination (see [the paragraph](#_Structure_of_Matlab) below).

If *saveIntermMat = false*, then only “output.mat” is saved, if *saveIntermMat = true*, then both “output.mat” and “intermediate.mat” are saved periodically.

*Parameter 14 is used only if saveBackupMats = true*

1. **backupPeriodIter**

The parameter specifies how often backup files should be saved. The saving is done once per “backupPeriodIter” iterations.

1. **backgroundMode**

The parameter specifies whether to run HPC kernel in background mode.

true — HPC kernel is run in background mode (MATLAB can be idle).

false — HPC kernel is run in foreground mode (MATLAB is busy).

The key difference between foreground and background modes is that the second mode does not require persistent connection between MATLAB host program and HPC kernel at time of the whole simulation session. In the first mode HPC kernel dumps information about current progress into console and MATLAB host shows this output to user as is. In the second case HPC kernel dumps information about current progress into status file from time to time and MATLAB host picks it up periodically, analyses and shows current progress to user.

*Parameters 16-18 are used only if backgroundMode = true*

1. **c4sPeriodSec**

The parameter specifies how often MATLAB host should check the status of HPC kernel. The status check means that the host program determines if HPC kernel is running and if so, what the current iteration number is and what the duration of iteration is. The status is checked once per c4sPeriodSec seconds. (c4s = Check For Status.)

1. **c4cPeriodIter**

The parameter specifies how often HPC kernel should check for a command from MATLAB host. There are two commands: terminate, dump snapshot (see [the paragraph](#_Structure_of_Matlab) below). The check is done once per c4cPeriodIter iterations. (c4c = Check For Command.)

1. **c4ePeriodSec**

The parameter specifies how often MATLAB host should check if HPC kernel has executed the requested command. There are two commands: terminate, dump snapshot (see [the paragraph](#_Structure_of_Matlab) below). The check is done once per c4ePeriodSec seconds. (c4e = Check For Execution.)

*Parameters 19-20 are used only if simulation is running on a cluster*

1. **idleMaster**

The parameter specifies whether the master node of the cluster idles.

true — all slave nodes do a job, the master node idles.

false — all slave nodes and the master node do a job.

1. **memPerNodeLimit**

The parameter specifies the limit of physical memory usage per cluster node (in megabytes). The simulation will not start if the limit is exceeded. The estimate of required physical memory depends on numbers of neurons, ring networks parameters, types of 4 synaptic conductance matrices, the mode of presynaptic voltage history matrices, the number of processes and on whether single-precision or double-precision arithmetics is used. The equations used for estimation of required physical memory are in [this](#_Memory_requirements) paragraph.

*Parameter 21 is used only if remoteHPC = true*

1. **zipMatFiles**

The parameter specifies whether input/output MAT-files should be zipped before and unzipped after transferring between local machine and head node of the cluster.

true — files are compressed before and decompressed after transferring.

false — files are transferred without compression.

## Changing system of equations

The system of equations is defined by the following functions: “h\_e\_inf”, “h\_i\_inf”, “m\_e\_inf”, “m\_i\_inf”, “n\_e\_inf”, “n\_i\_inf”, “tau\_h\_e”, “tau\_h\_i”, “tau\_n\_e”, “tau\_n\_i”.

Each function is defined in a separate CPP file under the following directory:

“/home/reviewer/gs/worker”.

The exception is for functions “m\_e\_inf” and “m\_i\_inf”. They are in both C++ files (“m\_e\_inf.cpp” and “m\_i\_inf.cpp”) and MATLAB files (“m\_e\_inf.m” and “m\_i\_inf.m”). If any changes, make sure that the code is identical in corresponding files.

When you are changing any functions, follow these rules:

1. Make sure that a function does not encounter uncertainty for some specific values of input argument.

For example, function “m\_e\_inf” encounters uncertainty 0 / 0 in the cases when input argument v equals –54 and –27.

You must provide special handling of the cases because otherwise the function can return NaN (Not-A-Number) at some moment of simulation making all further process spoiled.

1. Avoid usage of types float or double explicitly, use template type T instead.

Each function is compiled in two versions: T = float and T = double.

The version used at time of simulation depends on value of variable “useSPA” specified in file “HpcParams.m”.

Do not specify rational constants as 1.2 or 3.4 because they would have double type, specify them as T(1.2) and T(3.4) instead.

However, integer constants can be specified without indication of type T.

1. Avoid division of integer numbers. For example, defining variable x as

T x = T(0.5) + 1 / 3;

you will actually have x equal T(0.5). The reason is that quotient of integer numbers is rounded to integer.

The correct forms are:

T x = T(0.5) + T(1) / 3;

T x = T(0.5) + 1 / T(3);

1. Do not forget to compile HPC kernel running script “build\_Linux\_RELEASE.sh” living in the same directory.

If you added some new functions, make sure that they are specified with “inline” keyword and listed in files “GammaSimulator.h” and “GammaSimulator.cpp”.

## Choosing and initialization of random number generators

Random number generators are used in three places.

1. The MATLAB host program uses one to prepare input data for the simulation.

The generator is initialized with the seed specified by variable “seed” in file “PrepareInputData.m”.

1. If “scmType\_XY” is equal to “KrnDense”, “KrnSparse”, “KrnOneBit” or “KrnInPlace”, then the C++ worker program uses one generator per thread to initialize local chunks of distributed matrices “g\_ee”, “g\_ei”, “g\_ie”, and “g\_ii”. Each thread must have random number generator initialized with unique seed. If “scmType\_XY” is equal to “KrnInPlace”, then the matrices are generated SAME on different iterations.

The seeds for all threads of all processes are collected into one-dimensional array “scmSeeds” defined in file “PrepareInputData.m”. This array is saved to input MAT-file with other data. Each value of a seed, including positive values and negative values, corresponds to unique state of random number generator.

The requirements for “scmSeeds” are as follows:

a) “scmSeeds” must be integer vector of length not less than np **·** nt **·** 4;

b) “scmSeeds” must contain values in range [-2147483648, 2147483647] except 0;

c) “scmSeeds” must not contain equal values;

d) “scmSeeds” must not overlap with “uSeeds” (see below).

1. The C++ part of the software uses one generator per thread to initialize the local chunks of distributed vectors “u\_e” and “u\_i” in the file “DoSimulation.cpp”. Each thread must have random number generator initialized with a unique seed. The DIFFERENT vectors are generated on different iterations.

The seeds for all threads of all processes are collected into one-dimensional array “uSeeds” defined in the file “START\_GammaSimulator.m.” This array is saved to the input MAT-file with other data. Each value of a seed, including positive values and negative values, corresponds to unique state of random number generator.

The requirements for “uSeeds” are as follows:

a) “uSeeds” must be integer vector of length not less than np **·** nt;

b) “uSeeds” must contain values in range [-2147483648, 2147483647] except 0;

c) “uSeeds” must not contain equal values;

d) if “scmType\_XY” is equal to “KrnDense”, “KrnSparse”, “KrnOneBit” or “KrnInPlace”, then “uSeeds” must not overlap with “scmSeeds”.

It’s possible to choose quality of random number generators used in HPC kernel. Boolean variable “use32BitRng” defined in the file “HpcParams.m” switches between 32-bit and 64-bit random number generators.

32-bit generator std::mt19937 produces integer random numbers in the interval [0, 232). When these numbers are converted to floating-point ones and projected to the interval [0, 1), the step of the grid becomes equal approx. 2.3283 · 10–10. It's sufficient when useSPA = true because the step supported by the floats

eps(single(1)) = 1.1921 **·** 10–7.

But it's not sufficient when useSPA = false because the step supported by the doubles

eps(double(1)) = 2.2204 **·** 10–16.

64-bit generator std::mt19937\_64 produces integer random numbers in the interval [0, 264). When these numbers are converted to floating-point ones and projected to the interval [0, 1), the step of the grid becomes equal approx. 5.4210 **·** 10–20. It’s sufficient for reaching the limit of granularity of double-precision arithmetic.

## Changing parameters in the case of simulation continuation

If the parameter “saveIntermMat” is specified as true, then HPC kernel saves all necessary data so that simulation can be continued after stopping. In that case you cannot change any parameters of simulation; it will continue with the same parameters independently on any changes made in files “HpcParams.m” and “ModelParams.m”. The only exception is for the following three parameters:

1. np — the number of processes;
2. nt — the number of threads per process;
3. idleMaster — the flag indicating whether the master node idles.

You can change these parameters, and this change will be applied in the next simulation session. The only restriction is that the product np **·** nt must be the same as before.

# Model restrictions, memory requirements and performance

## Bit synaptic conductance matrices

Synaptic conductance matrices have a bit structure if *scmType\_XY = KrnOneBit*. They provide compromise between the cases “performance is low, but big models are allowed” (*scmType\_XY = KrnInPlace*) and “performance is high, but big models are not allowed” (*scmType\_XY = KrnDense*). The principal limitation of the 1-bit mode is that a synaptic conductance matrix can be populated with only two different values (zero and some other value).

In order to achieve the best performance, the number of rows in 1-bit element matrix should be evenly divisible by 64. If this is not the case, then the number of rows (and the number of neurons of a type) is rounded to the nearest less integer that is evenly divisible by the factor.

Let’s consider the following example:

num\_e = 3000; num\_i = 5000;

scmType\_ee = KrnSparse; scmType\_ei = KrnOneBit;

scmType\_ie = KrnSparse; scmType\_ii = KrnSparse.

The type “scmType\_ei” requires rounding of number of neurons (since it is equal to “KrnOneBit”), while three other types do not require any rounding. The number of rows in matrix “g\_ei”, i.e. “num\_e”, should be evenly divisible by 64. As a result, the numbers of neurons will be rounded as follows:

num\_e = 2944; num\_i = 5000.

## Memory allocated on a cluster node

The memory requirements are checked by the function that lives in the file “host\OtherUtils\CheckMemReq.m”. If memory allocated by four synaptic conductance matrices and four matrices of presynaptic voltage history exceeds the threshold “memPerNodeLimit”, then the function prevents simulation launching to avoid the memory deficit on cluster nodes.

Each type of synaptic conductance matrix specified by “scmType\_XY” parameter has specific memory requirements checked by the function that lives in the file “host\OtherUtils\CountMatrixMemReq.m”. The requirements for physical memory per cluster node (in MB) for synaptic conductance matrices are estimated as follows:

|  |  |
| --- | --- |
| **Synaptic conductance matrix type** | **Memory allocated for the matrix on a cluster node (in MB)** |
| AllZeros  AllEqual  KrnInPlace | **reqMemPerNode = 0**.  This is a rough estimate that takes into scope that the matrices are not stored in memory explicitly. |
| HstDense | **reqMemPerNode = numElem · elemSize · (np + 1) / np / 220**  with “numElem” being the number of elements in the matrix,  “elemSize” being the size of an element. The size is equal 4 if *useSPA = true* and 8 otherwise.  This estimate takes into scope that the most memory is allocated on the master node after loading of the matrix from MAT-file and scattering. |
| HstSparse | **reqMemPerNode = (f(intSize) + f(intSize / 2) / np) / 220**  with **f(size) = numElem · (elemSize + size) + (numCols + 1) · size**,  “numElem” being the rough estimate of the number of nonzero elements in the matrix which depends on the height and the standard deviation of the bell:  **numElem = round(w\_max · (exp(-A2) - 1 + A · sqrt(π) · erf(A)) / B),**  **B = A2 / (numRows - 1) / (numCols - 1),**  **A = sqrt(0.5) · (max(numRows, numCols) - 1) / sigma**,  “elemSize” being equal to 8 (MATLAB does not support single-precision sparse arrays),  “intSize” being equal to 8 (MATLAB does not support 4-bit integer sparse arrays).  This estimate takes into scope that the most memory is allocated on the master node after loading of the matrix from MAT-file and scattering. |
| KrnDense | if *useHC = true* and *saveIntermMat = true* or *showSCM = true*  **reqMemPerNode = numElem · elemSize · (np + 1) / np / 220**  else  **reqMemPerNode = numElem · elemSize / np / 220**.  The estimate is similar to that one for “HstDense”, but if we don’t need to gather the matrix on the master node for writing to MAT-file this estimate is (np + 1) times lower. |
| KrnSparse | **reqMemPerNode = (numElem · (elemSize + intSize) + (numCols + 1) · intSize) / np / 220**.  with “numElem” being similar to that one for “HstSparse”,  “intSize” being equal 4,  “elemSize” being the size of an element. The size is equal 4 if *useSPA = true* and 8 otherwise. |
| KrnOneBit | **reqMemPerNode = numElem / blockSize · intSize / np / 220**  with “numElem” being the number of elements in the matrix,  “blockSize” being 64,  “intSize” being 8. |

When scalability test is run, the worst value for number of processes “minNP” is substituted instead of “np”.

Notice that mode “KrnOneBit” requires the number of rows in the matrix to be evenly divisible by the 64. If this is not the case, rounding to the nearest lower integer value that fits the requirement is performed. As a result, allocated memory becomes lower (see [previous paragraph](#_1-bit_synaptic_conductance)).

The requirements for physical memory per cluster node (in MB) for presynaptic voltage history matrices depend on value of variable “distMatPVH” (see the paragraph [The file "HpcParams.m"](file:///C:\Users\lsavtchenko\AppData\Roaming\Skype\My%20Skype%20Received%20Files\HpcParams.m#_The_file_)) and are estimated as follows:

|  |  |
| --- | --- |
| **distMatPVH** | **Memory allocated for the matrix on a cluster node (in MB)** |
| false | **reqMemPerNode = numElem · elemSize / 220**  with “numElem” being the number of elements in the matrix,  “elemSize” being the size of an element. The size is equal 4 if *useSPA = true* and 8 otherwise. |
| true | **reqMemPerNode = numElem · elemSize / np / 220**  The estimate is similar to that one for *distMatPVH = false*, but np times lower |

Notice that if *saveIntermMat = true*, we have to gather presynaptic voltage history matrices for writing them to intermediate MAT-file, i.e. the memory for whole matrix has to be allocated on master node. Therefore this value of variable “saceIntermMat” is forbidden when *distMatPVH = true* to avoid unpractical usage of memory.

## Limitations of the number of neurons caused by memory requirements

Let’s consider the following example:

num\_e = 2 · num\_i; radius\_e = 250; radius\_i = 200; v = 0.1; sclModel = BSD; useHC = false;

w\_ee\_max = 2; w\_ii\_max = 0.8; w\_ei\_max = 0.9; w\_ie\_max = 0.3;

sigma\_ee = 100; sigma\_ei = 75; sigma\_ie = 75; sigma\_ii = 50;

np = 9; useSPA = true; saveIntermMat = false; memPerNodeLimit = 1700.

The maximum number of neurons with which simulation can be launched is estimated as follows:

|  |  |  |
| --- | --- | --- |
| **scmType\_XY** | **Maximum number of neurons** | |
| **distMatPVH = true** | **distMatPVH = false** |
| KrnDense | num\_e = 29000; num\_i = 14500 | num\_e = 8800; num\_i = 4400 |
| KrnSparse | num\_e = 39600; num\_i = 19800 | num\_e = 8800; num\_i = 4400 |
| KrnOneBit | num\_e = 39400; num\_i = 19700 | num\_e = 8800; num\_i = 4400 |
| KrnInPlace | num\_e = 40000; num\_i = 20000 | num\_e = 9000; num\_i = 4500 |

## Performance of ARACHNE

The main parameters that determine ARACHNE performance are the following:

1. the number of processes “np”;
2. the number of threads “nt”;
3. the numbers of neurons “num\_e” and “num\_i”;
4. the parameters of ring networks “radius\_e”, “radius\_i” and “v”;
5. the types of synaptic conductance matrices “scmType\_XY” with “XY” being “ee”, “ei”, “ie”, “ii”;
6. the mode of presynaptic voltage history matrices “distMatPVH”;
7. the variable “useHC” which specifies whether to correct synaptic conductance matrices according to Hebbian theory;
8. the maximum heights of corresponding bells “w\_XY\_max” (for sparse matrix only);
9. the standard deviations of corresponding normal distributions “sigma\_XY” (for sparse matrix only).

In general case, the most time-consuming operation performed on each iteration is computation of the actual current. This operation has quadratic complexity while other operations have linear complexity. The current is computed 8 times on each iteration. For each matrix of synaptic conductance (“g\_ee”, “g\_ei”, “g\_ie”, “g\_ii”) the operation is provided twice.

Let’s consider complexity of current computation for a matrix “g\_XY” of size “num\_X” by “num\_Y”. Denote *N1 = num\_X + num\_Y* — the sum of height and width of the matrix, *N2 = num\_X · num\_Y* — the number of elements in the matrix, *N3* — the number of nonzero elements in sparse matrix which depends on “num\_X”, “num\_Y”, “w\_XY\_max” and “sigma\_XY”, *n = np · nt* — the total number of workers. Then complexity of current computation for the matrix “g\_XY” is estimated as follows:

| **Synaptic conductance matrix type** | **Complexity** |
| --- | --- |
| AllZeros | O(num\_Y / n) |
| AllEqual | O(N1 / 2 / n) |
| HstDense  KrnDense  KrnOneBit  KrnInPlace | O(N2 / 2 / n) |
| HstSparse  KrnSparse | if *distMatPVH = false*  O(N3 / n)  if *distMatPVH = true*  O(N3 · num\_X / 2 / n) |

That means that doubling of the number of workers and preserving all other parameters unchanged brings halving of calculation time. The statement is true with the following remarks:

1. When ARACHNE is running with *distMatPVH = true*, this statement is true only if number of neurons and processes is rather little. The cause of this effect is that communication between processes on each iteration becomes huge when the number of neurons is rather big. Usually with little number of neurons and processes the mode *distMatPVH = false* with better performance can be used. If there is a memory deficit and ARACHNE can be running only with *distMatPVH = true*, the number of processes must be as little as it is possible.
2. When ARACHNE is running with *np > 1*, there is the overhead due to communication between processes. As a result, performance for *np = 2* becomes approximately twice better than performance for *np = 1* only when the number of neurons is large enough to compensate the overhead.
3. When ARACHNE is running with *nt > 1*, the overhead due to communication between threads is much less than overhead due to communication between processes. As a result, performance for *nt = 2* becomes approximately twice better than performance for *nt = 1* for much less number of neurons, than it was necessary in the point 1.
4. The “O” symbol used in the table denotes linear dependence on the argument, but does not state the slope coefficient. The slope depends on synaptic conductance matrix type. For example, the slope for “KrnInPlace” matrix type is higher that the slope for “KrnDense” matrix type. (The product is computed slower for the first type and faster for the second type.) At the same time, the slope coefficients are equal for the following types of synaptic conductance matrices:
5. “HstDense” and “KrnDense”;
6. “HstSparse” and “KrnSparse”.

## Scalability curves for different types of matrices

Let’s consider the following example:

num\_e = 8700; num\_i = 4350; radius\_e = 250; radius\_i = 200; v = 0.1;

sclModel = BSD; useHC = false; t\_final = 0.2;

w\_ee\_max = 2; w\_ii\_max = 0.8; w\_ei\_max = 0.9; w\_ie\_max = 0.3;

sigma\_ee = 1000; sigma\_ei = 750; sigma\_ie = 750; sigma\_ii = 500;

scalTest = true; np = 9; nt = 4; minNP = 8; useSPA = true; use32BitRng = true;

memPerNodeLimit = 1700; stabAnalysis = false; saveIntermMat = false; saveBackupMats = false.

When *distMatPVH = false* the simulation scales, but the scalability is far away from the ideal. The cause of this effect is memory deficit which make impossible to consider enough neurons to show real scalability.

|  |  |
| --- | --- |
|  |  |
| Picture 4.1. *scmType\_XY = KrnDense* | Picture 4.2. *scmType\_XY = KrnSparse* |
|  |  |
| Picture 4.3. *scmType\_XY = KrnOneBit* | Picture 4.4. *scmType\_XY = KrnInPlace* |

When *distMatPVH = true* there is no scalability in most cases because of active communication between processes. The simulation scales only with little number of neurons and processes, i. e. when communication between processes isn’t very sizeable, but in this case the option *distMatPVH = false* is better to use.

# Structure of MATLAB host program

The main file of the host program is “**START\_GammaSimulator.m”**. It’s used for

1. starting of simulation from scratch;
2. attaching to and monitoring of simulation running in background at the moment;
3. continuing previous simulation session or continuing simulation after a crash and a recovery;
4. running scalability test;
5. grabbing results of previous simulation session and showing them to user.

The next three scripts are used for servicing of background simulation mode:

1. **“SCRIPT\_GetSnapshot.m”** — this script does the request to HPC kernel to dump current results of simulation. After the dumping, the results file “output.mat” is taken by the host program (downloaded if *remoteHPC = true* or copied otherwise) and analysed. Then the results are shown to user graphically. When HPC kernel creates a snapshot, it also creates a backup point that can be used for progress recovery.
2. **“SCRIPT\_TerminateBackgroundProcess.m”** — this script does the request to HPC kernel to save current results and terminate.
3. **‘SCRIPT\_KillBackgroundProcess.m”** — this scripts terminates all HPC kernel processes forcibly. Current progress is lost, but it’s possible to recover some old progress from backup file(s) if the variable “saveBackupMats” was equal “true”.

Notice that the scripts 1 and 2 described above execute asynchronous operations between host and kernel, therefore, require some time. They are controlled by variables c4cPeriodIter and c2ePeriodIter defined in the file “HpcParams.m”.

The next two scripts are used for other servicing of ARACHNE:

1. **“SCRIPT\_RecoverBackupProgress.m”** — this script recovers progress of simulation from the latest backup file(s). It’s worth in the case if simulation was stopped unexpectedly for some reason not depending on the program (e.g. impact of hardware, network, other software etc.). In order to make ARACHNE save backup file(s) periodically, specify *saveBackupMats = true* in the file “HpcParams.m”. Both files “output.mat” and “intermediate.mat” are subjects of recovery, but the latter file can be recovered only if simulator was set up to save the file (i.e. *saveIntermMat = true*). There are two backup storages used by HPC kernel in turn and backup storage pointer file that points to the storage that will be used for saving next time. Such approach makes it possible to recover even in the case if crash happens at the moment of backup file(s) saving.
2. **“SCRIPT\_CleanUp.m”** — this script cleans up HPC kernel I/O directories (see [the paragraph](#_I/O_directories) below). After running of the script, it’s impossible to get results of the latest simulation, continue or recover progress. All the data are deleted. Simulation can be run only from scratch.

The script **“UTILITY\_GetMaxModelSize.m”** is used to get the maximum size of model which meets memory requirements. It calls “ModelParams.m” and “HpcParams.m” to initialize all required parameters except of “num\_e” and “num\_i”, asks user to enter the ratio “num\_i / num\_e” and computes the maximum number of neurons for adjusted model and HPC parameters.

# Structure of HPC kernel

## Source code

|  |  |
| --- | --- |
| **File** | **Description** |
| main.cpp | The file contains ARACHNE HPC kernel entry point function “main”. It reads variables “useSPA” and “use32BitRng” from input MAT-file and calls templated function “main\_templated” with appropriate type arguments. The templated function calls three modules to read all other stuff of the MAT-file, run the simulation itself and save results to output MAT-file(s). |
| MatFileIOUtils.h  MatFileIOUtils.cpp | Utility functions to read scalars, vectors and matrices from input MAT-file and broadcast or scatter them among processes.  Utility functions to save scalars, vectors and matrices to output MAT-files. |
| GammaSimulator.h  GammaSimulator.cpp | Templated class containing all data and functions comprising internal logic of33 simulation. |
| ReadAllocateWrite.cpp | The modules of ARACHNE responsible for reading of all variables from input MAT-file, allocating memory for them and other internal objects, writing of simulation results to output MAT-files. |
| DoSimulation.cpp | The time evolution cycle module. |
| UpdateIdxTNumSpikes.cpp | Selection of cells spiked at current time step. Update of the arrays that track spiked cells. |
| h\_e\_inf.cpp  h\_i\_inf.cpp  m\_e\_inf.cpp  m\_i\_inf.cpp  n\_e\_inf.cpp  n\_i\_inf.cpp  tau\_h\_e.cpp  tau\_h\_i.cpp  tau\_n\_e.cpp  tau\_n\_i.cpp | The functions defining system of equations. In the most they migrated from the original software without changes. The only new feature is the control of uncertainties added to avoid spoiling of simulation in the case if a function encounters division 0 / 0. |
| ExternalDrives.cpp | The functions defining external drives to E- and I-cells (migrated from file “params.m”). |
| LocalVector.h  LocalVector.cpp | Templated class representing local vector being analogue of one-dimensional dense array in MATLAB. |
| DistVector.h  DistVector.cpp | Templated class representing distributed vector, i.e. the vector with data evenly divided among processes. |
| ScmType.h | Enumerable that specifies the type of synaptic conductance matrix. |
| LocalMatrix.h  LocalMatrix.cpp | Templated class representing local dense matrix being analogue of two-dimensional dense array in MATLAB. |
| LocalSparseMatrix.h  LocalSparseMatrix.cpp | Templated class representing local sparse matrix being analogue of two-dimensional sparse array in MATLAB. The difference is that MATLAB does not support single-precision sparse arrays, while this class does support. |
| DistMatrixBase.h | Templated abstract base class for all distributed matrix classes: “ZeroDistMatrix”, “ConstDistMatrix”, “DistMatrix”, “DistSparseMatrix”, “OneBitDistMatrix” and “InPlaceDistMatrix”. The class represents a matrix with columns evenly divided among all processes (i.e. each process contains all the rows, but only some number of columns). All the classes derived from this class must implement the method that computes the current on actual iteration. |
| ZeroDistMatrix.h  ZeroDistMatrix.cpp | Templated class representing distributed matrix with all elements equal zero. The class is used if *scmType\_XY = AllZeros*. |
| ConstDistMatrix.h  ConstDistMatrix.cpp | Templated class representing distributed matrix with all equal elements. The value of all elements equal to is stored in an object of the class. The class is used if *scmType\_XY = AllEqual*. |
| DistMatrix.h  DistMatrix.cpp | Templated class representing distributed dense matrix with all elements stored in floating-point format. The class is used if *scmType\_XY = HstDense* or *scmType\_XY = KrnDense*. |
| DistSparseMatrix.h  DistSparseMatrix.cpp | Templated class representing distributed sparse matrix with all elements stored in floating-point format. The class is used if *scmType\_XY = HstSparse* or *scmType\_XY = KrnSparse*. |
| OneBitDistMatrix.h  OneBitDistMatrix.cpp | Templated class representing distributed dense matrix populated with only two different values. Each matrix element is stored in one bit of memory. Unset bit corresponds to zero matrix element, set bit corresponds to some value. (The value is a scalar contained in the class object.) The class is used if *scmType\_XY = KrnOneBit*. |
| InPlaceDistMatrix.h  InPlaceDistMatrix.cpp | Templated class representing distributed dense matrix generated in place. The matrix allocates just a little amount of physical memory; it’s reconstructed as a sequence of elements many times at time of simulation. The class object contains random number generator seeds to make each local chunk of the matrix re-generated same. The class is used if *scmType\_XY = KrnInPlace*. |
| DistMatrixFactory.h  DistMatrixFactory.cpp | Factory function that creates distributed synaptic conductance matrices with parameters read from the file "input.mat". If “scmType\_XY” is equal to “KrnDense”, “KrnSparse”, “KrnOneBit” or “KrnInPlace”, then each matrix is initialized with other set of seeds for random number generators. |
| ElementwiseOp1.cpp  ElementwiseOp2.cpp  ElementwiseOp3.cpp  ElementwiseOp4.cpp | Element-wise operations being part of mathematical core of ARACHNE. Each operation has many input vectors and many output vectors. The vectors are processed in element-wise manner by different processes/threads independently. |
| ComputeSynCoef.cpp | Method that computes synchronization coefficient between neuron spikes. |
| DistEnv.h  DistEnv.cpp | Distributed environment variable and utility function staff. |
| FixCurrentProgress.cpp | Method that prints overall progress and refreshes status file in background simulation mode.  Method that saves backup files. |
| OtherFileIOUtils.h  OtherFileIOUtils.cpp | I/O file utility functions that are not related to reading from and writing to MAT-files. They provide support of background simulation mode and backup saving of the progress. |
| DistributionWrapper.h  DistributionWrapper.cpp | Wrapper of the random number generator facade class std::uniform\_real\_distribution<T>. The wrapper itself plays minor technical role making it possible to use the generator in particular software. |
| GetTypeTagUtils.h | Templated utility functions that return MATLAB data type tag and MPI data type tag given template argument. |
| MathUtils.h  MathUtils.cpp | Various mathematical functions used in tests. |

Notice that functions “m\_e\_inf” and “m\_i\_inf” live not only in C++, but also in MATLAB files “m\_e\_inf.m” and “m\_i\_inf.m”. If any changes, make sure that the code is identical.

## I/O directories

All files HPC kernel works with live in the directory “worker/iofiles.” The directory contains some subdirectories listed below:

1. “iofiles/host-kernel” — this directory is used to store all files MATLAB host passes to HPC kernel. The files are:
2. “input.mat” — input data for simulation,
3. “terminate” — empty file signalling about termination request,
4. “snapshot” — empty file signalling about snapshot request.
5. “iofiles/kernel-host” — this directory is used to store all files HPC kernel passes to MATLAB host. The files are:
6. “output.mat” — the file with results of simulation,
7. empty file with name pattern “iter %i - %i; %g sec” or “iter %i (%i - %i); %g sec” used to report current progress of HPC kernel running in background mode.
8. “iofiles/kernel-host/snapshot”— this directory is used to store the snapshot file “output.mat” dumped by HPC kernel in background mode per user’s request.
9. “iofiles/kernel-kernel” — this directory is used to store the file “intermediate.mat” saved by HPC kernel and loaded afterwards to continue previous simulation session (the files “input.mat” and “output.mat” are read in that case as well).
10. “iofiles/backup” — this directory is used to store backup data. There are two backup storage directories: “iofiles/backup/backup-1” and “iofiles/backup/backup-2” used by HPC kernel in turn. Each backup storage directory contains two subdirectories: “kernel-host” for the file “output.mat” and “kernel-kernel” for the file “intermediate.mat.” There is empty pointer file in the directory “iofiles/backup” that points to the backup storage directory that will be used for the next backup saving. The name of the file “1” or “2” indicates the storage.

# Passing new parameters from host (MATLAB) to workers (C++)

## Passing a new scalar to an equation

Let’s consider the case when you need to make some equation dependent on some new floating-point scalar variable defined in MATLAB.

For example, you want to make function “tau\_n\_e” defined in file “tau\_n\_e.cpp” dependent on variable “myScalar” defined in MATLAB code.

1. Open file “PrepareInputData.m” and find “if useSPA” statement. Add your variable to the body of the statement as it was done for other variables.
2. Find “input = ...” statement and add the name “myScalar” to the list using single quotes (notice that MATLAB syntax uses single quotes ′ instead of double quotes ″ to denote string literals).
3. Log in to the tuxmaster node and open the file

“/home/reviewer/gs/worker/GammaSimulator.h”.

Find the place where the following variables “dt”, “dt05”, “v\_rev\_e”, … are declared and add the declaration of new variable “myScalar” in the same way.

1. Open file “/home/reviewer/gs/worker/ReadAllocateWrite.cpp” and find the place where the variables “dt”, “v\_rev\_e”, “v\_rev\_i”, … are read from MAT-file.

Add one more line for the new variable as follows:

myScalar = ReadCheckBroadcastScalar<T>("myScalar").

1. Open file “/home/reviewer/gs/worker/tau\_n\_e.cpp” and utilize the new variable as you need. For example, you can replace line

T beta\_n = T(0.5) **·** exp(–(v + 57) / 40);

with the following line:

T beta\_n = myScalar **·** exp(–(v + 57) / 40).

1. Run script “/home/reviewer/gs/worker/build\_Linux\_RELEASE.sh” to recompile the program.

## Passing a new vector to an element-wise operation

Let’s consider the case when you need to make some element-wise operation dependent on some new floating-point vector defined in MATLAB.

### The case of scattering of the vector

The scattering means that the vector read from input MAT-file on the master process is divided into parts so that each process receives only one part of the vector.

Let’s suppose that you want to make function “ElementwiseOperation1” defined in file “ElementwiseOp1.cpp” dependent on vector “myVector” of length “num\_e” defined in MATLAB code.

1. Open file “PrepareInputData.m” and find “if useSPA” statement. Add your vector to the body of the statement as it was done for other variables.
2. Find “input = ...” statement and add the name “myVector” to the list using single quotes.
3. Log in to tuxmaster node and open the file

“/home/reviewer/gs/worker/GammaSimulator.h”.

Find the place where the following vectors “v\_e”, “n\_e”, “m\_e”, … are declared and add the declaration of new vector “myVector” in the same way.

1. Open file “/home/reviewer/gs/worker/ReadAllocateWrite.cpp” and find the place where the vectors “v\_e”, “n\_e”, “m\_e”, … are read from MAT-file.

Add one more line for the new vector as follows:

myVector = ReadCheckScatterVector<T>("myVector", num\_e);

1. Open file “/home/reviewer/gs/worker/ElementwiseOp1.cpp” and utilize the new vector as you need. Use “myVector[idx]” to read an element of the vector in the cycle.
2. Run script “/home/reviewer/gs/worker/build\_Linux\_RELEASE.sh” to recompile the program.

### The case of broadcasting of the vector

The broadcasting means that the vector read from input MAT-file on master process is cloned among all processes. In general, passing a vector in such way from MATLAB to HPC kernel is similar to the case of scattering of the vector described above. The only differences are as follows:

1. The vector should be read form MAT-file with help of other method:

myVector = ReadCheckBroadcastVector<T>("myVector", num\_e);

1. Access to elements of broadcasted vectors in element-wise operations should not be done by the same indices as for scattered vectors. The reason is as follows. Length of local portion of scattered vector depends on number of processes while local length of broadcasted vector does not depend on number of processes. As a result, access by “myVector[idx]” may cause out-of-range error if “idx” is the same index variable used to access scattered vectors of length “num\_e”.