# Parallelizing a Neural Network Model of Systems Memory Consolidation and Reconsolidation

Grace Dessert, Minhaj Hussain

Duke University, Department of Biomedical Engineering

BME590L - Comp. Foundations of Biomedical Simulation

Prof. Amanda Randles

04/19/2022

## Introduction

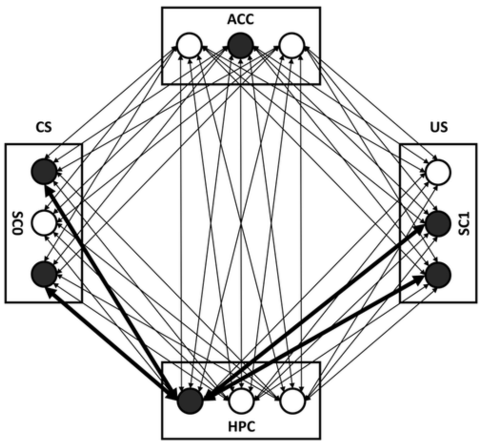
We present two methods for parallelizing a neural network model of systems memory consolidation and reconsolidation, originally described in (Helfer & Shultz, 2020), and contrast performance in deployment up to 256 tasks across 4 nodes on Stampede2 (Texas Advanced Computing Center (TACC), The University of Texas at Austin).

Memory consolidation is the process of transforming recent memories, which depends on the hippocampus (HPC), into long-term memories that depend on the neocortex. When these memories are activated again, they can become HPC-dependent again during memory reconsolidation. On a systems level, these processes are poorly understood, so a computational model could help elucidate the functioning of these systems in their normal state as well as in altered states such as with lesions or protein synthesis inhibitor (PSI) infusion.

While biological neural networks consist of millions of neurons and communicate with action potentials and synapses, we can model the systems-level of these processes from an abstract level using artificial neural networks. Helfer and Shultz's artificial neural network model works well to simulate memory consolidation and reconsolidation on a small scale, but it is unable to scale to larger network simulations. Larger simulations could improve the accuracy of our memory models, allowing us to train more stimulus pairs on the same network, recover larger memories, add additional brain regions, and improve the resolution of the effects we see given different interventions. With the current implementation, running a standard simulation could easily take hours if we scaled the number of neurons by even two orders of magnitude compared to the original implementation (which uses a heavily restricted neural population size of 25 neurons per modeled brain region). Moreover, larger network sizes would quickly exceed the memory capacity of standard desktop hardware.

## Original Model Implementation

The serial code was downloaded from [ModelDB: A computational model of systems memory consolidation and reconsolidation (Helfer & Shultz 2019) (yale.edu)]().



Developed by Helfer and Shultz (2019), this artificial neural network simulates the memory consolidation and reconsolidation processes using dynamics based on the processes of long-term-potentiation (LTP). There are four regions in this network (the anterior cingulate cortex (ACC), hippocampus (HPC), and two sensory cortex areas (SC0 & SC1)), each modeled with 25 neurons with connections described by the diagram to the right.

Each synapse (unidirectional connection between 'units', i.e. model neurons) in this model has four parameters describing the state of LTP at that synapse. Synapses exist between all neurons in both directions for connected layers, however there are no intra-layer synaptic connections and no direct synaptic connections between the cortical regions SC0 and SC1. The network is trained on a set of conditioned (CS) and unconditioned stimulus (US) pairs, where the CS is presented at SC0 and the US at SC1, and the parameters of each synaptic connection are iteratively updated over the course of a simulation following either a Hebbian learning rule or AMPAR trafficking and decay. Memory is then defined as the ability of the network to recover a signal (state of the network) given a partial representation (i.e., recover the settled post-training state of SC1 when SC0 is clamped in the original US state). After applying different interventions, such as 'freezing' brain regions such that they no longer participate in the network, we can study the resulting effects on memory processes to both validate our systems model and understand more about how these systems work in the brain.

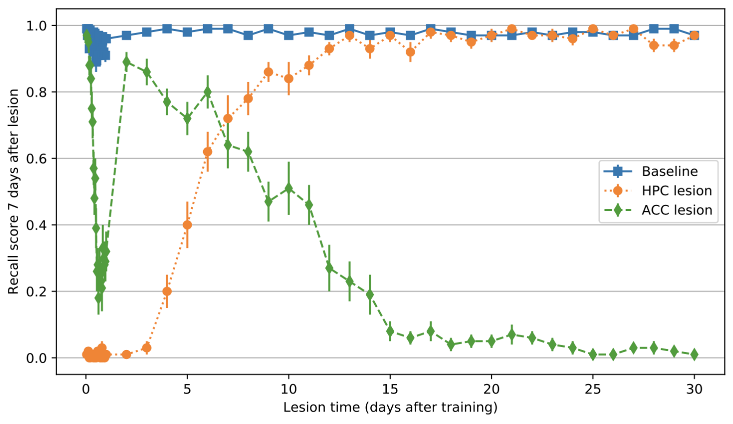


Figure 1. Consolidation window network simulation. The series indicate strength of recall over time with lesions to the HPC (green) and ACC (orange) at time of recall.

The dominant simulation in this model calculates the quality of system recall given lesions to the HPC and ACC across time. A lesion means that the connections to and from that region are disconnected and therefore that the region is unable to contribute to the reproduction of the US in SC1. A perfect reproduction of the US represents perfect memory and is given a recall score of 1. The output plot of this simulation shows the recall score as a function of time with 1) no intervention, 2) lesion to the HPC during the recall task, and 3) lesion to the ACC during the recall task. We see that in the baseline case, a near-perfect reproduction, which we think of as a memory, is always maintained. However, lesions to the HPC cause complete or severe loss of memory in the first few days and up to the first 10 days of the simulation. This is due to the integral role of the HPC in memory consolidation. We see that ACC lesions, on the other hand, cause loss of memory primarily after the consolidation phase, at which point the memory has become HPC-independent and ACC-dependent. This simulation of memory recall illustrates the roles of the HPC and ACC in memory consolidation and storage, and it is used throughout this analysis as the simulation of interest for parallel optimization and testing.

Figure 2 illustrates the memory layout and implementation model of the shared memory version of the code. Implemented in C++, every component of the network, i.e., individual neurons, and individual connections between pairs of neurons, as well as collections of these components, i.e., layers and tracts (collections of synapses between two interconnected layers) are implemented as instances of separate classes (`NsUnit`, `NsConnection`, `NsLayer`, `NsTract`).

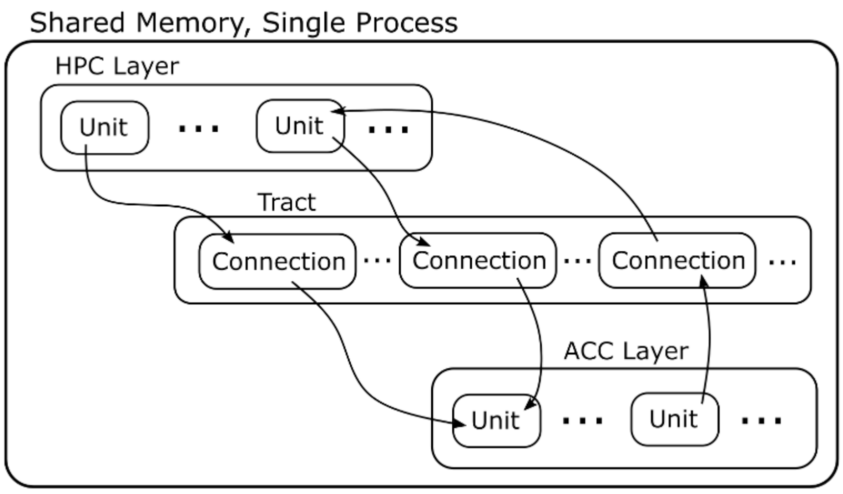


Figure 2. Memory layout schematic for serial implementation. Black arrows indicate pointers to locations in memory.

In the shared memory implementation, a `Unit` updates its activation (true or false) by iterating over a vector of pointers to all 'in' connections (i.e. `Connections` for which it is the postsynaptic cell) and performing a weighted sum over the synaptic strengths of all connections for which the presynaptic cells are active; a sigmoid nonlinearity is applied to this sum and the result is stochastically thresholded, returning `true` (active) or `false` (inactive). These boolean activation values are implemented as instance variables of the `Unit` class and during this update are accessed via a pointer to the `Unit` instance itself - this pointer is an instance variable of the `Connection` class. The synaptic strength is given by `(numCiAmpars + numCpAmpars) / maxPsdSize`, which are all instance variables of the `Connection` class of type `double`; these values are updated during Hebbian learning which induces changes in the number of AMPA receptors and size of the postsynaptic density to increase synaptic strength when both pre- and post- synaptic cells are active. The data dependencies are therefore apparent: the activation values of the model neurons and the parameters characterizing the model synaptic strengths.

### *Profiling Serial Code*

We profiled the serial code using the `valgrind` tool `callgrind` (results summarized in Figure 3). As expected, most time is spent in calculating the updated activations of the neurons in the model (`NsUnit::computeNewActivations`), followed by updating synaptic weights (`NsConnection::amparTrafficking`). These components of the model were therefore distributed in the parallel implementation, by instantiating disjoint subsets of all `Units` and `Connections` on each rank.

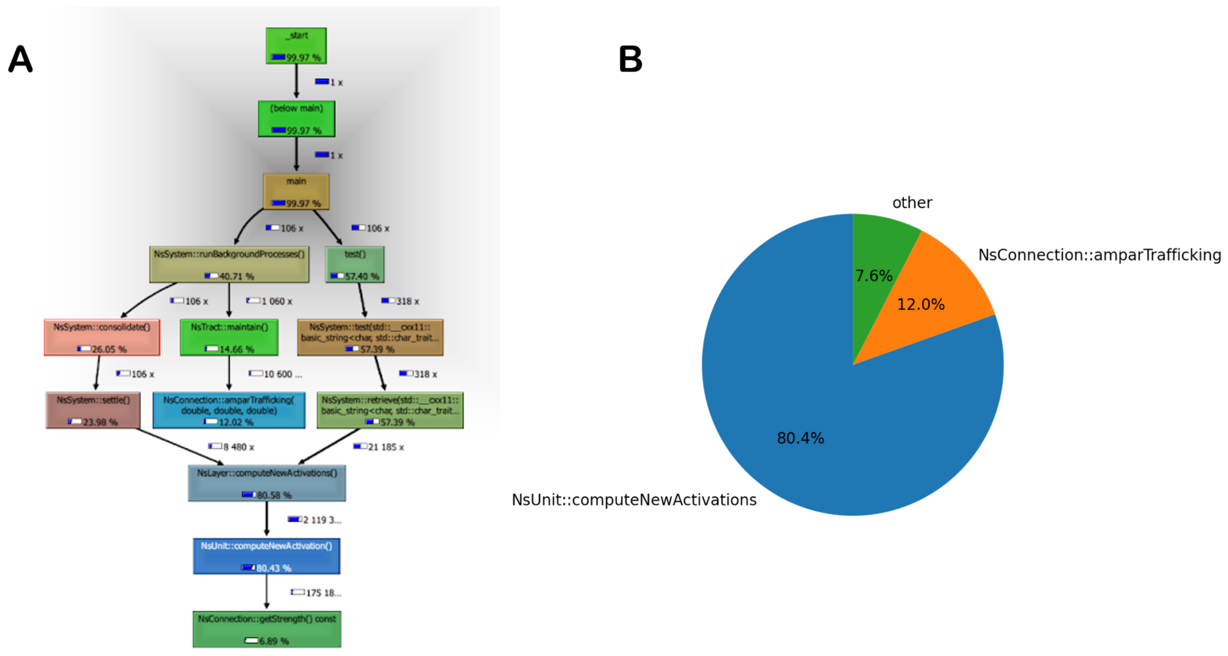


Figure 3. **A)** Call graph of serial implementation (generated using callgrind and kcachegrind). **B)** Summary pie chart of time spent within functions for serial implementation.

## Parallelization Schemes and Implementation

Standard biophysical models of spiking neural networks represent continuous state variables with ODEs and synaptic activity as *events*, characterized by their presynaptic origin and synaptic event time (i.e., the time at which a presynaptic spike occurred). This has the advantage that synaptic events can be synchronized across tasks in a distributed-memory system at intervals that greatly exceed the timestep used for integration of the ODEs, determined by the global minimum synaptic delay between any two cells on different ranks. In production biophysical network modeling software, this spike exchange between ranks is typically implemented using (fundamentally) an MPI_Allgather operation (Lytton et al., 2016).

Under the abstractions adopted in the Helfer and Shultz model, none of the neurons exhibit spiking behavior and no ODEs need to be solved; however the state (active or inactive, 1 or 0) of a neuron at a given timestep depends on the state of all presynaptic neurons with connections to that neuron from the previous timestep, and so these states must be exchanged between ranks at the end of every timestep. We implemented two approaches to achieve this: the ‘round-robin’ approach uses an MPI_Allgather call across the global communicator such that the states of all neurons are available on all ranks at the end of every timestep, and the ‘layer-based’ approach uses MPI_Iallgatherv across intergroup communicators to exchange activation data only across tracts between interconnected layers.

The class-based abstractions adopted by the original implementation and resulting data fragmentation are likely not optimal from a performance perspective due to cache inefficiency and the overhead of multiple pointer indirections to access data (see Discussion for further comment). However, we wanted to preserve the underlying code-base and make as few changes as possible to how the arithmetic logic was implemented while introducing parallelism with MPI. To this end, the only significant change to the original data structures that we implemented was with respect to how the `Unit` activation values were organized in memory, their data type, and how they were accessed. In both the round-robin and layer-based implementations, these data were moved from being `bool` instance variables of the `Unit` class to elements of contiguous `uint8_t` arrays (the MPI standard does not include a specific `bool` type), enabling us to use standard array-based MPI programming. In the round-robin case, this is a single global array (see Round-Robin Approach below), whereas in the layer-based approach these are separate layer-specific arrays (see Layer-Based Approach below).

## Round-Robin Approach

Both approaches are characterized in three fundamental ways: a graph partitioning policy (i.e., how `Units` and `Connections` are distributed between ranks), a data allocation policy (i.e., how the activation arrays are allocated), and a data exchange policy (i.e., how data finds its way onto the ranks where it's needed).

For the round-robin approach, we assigned every `Unit` a unique global identifier (`gid`), an integer in [0, totalUnits) where totalUnits is the total number of Units in the entire model. Units were then round-robin allocated (hence the approach name) to a rank if `gid % rank == 0`. All `Connections` were instantiated on the same rank as the *postsynaptic* `Unit` (as it is the postsynaptic `Unit` that must iterate through a local vector of input `Connections` in order to calculate its future state). Every rank was allocated a single contiguous global activations array to accommodate all Units in the entire model (i.e. `uint8_t [totalUnits]`). The partitioning scheme is illustrated in Figure 4.

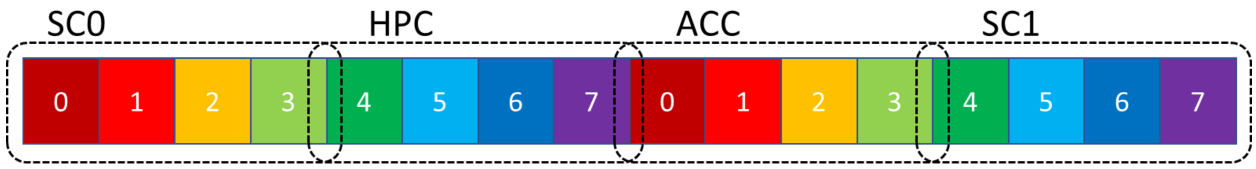


Figure 4. Illustration of the round-robin partitioning scheme for a hypothetical 16 Unit model (4 units per layer) running on 8 ranks. The numbers / colors indicate which rank 'owns' that element of the contiguous activations array (i.e. is responsible for updating that element prior to synchronization), corresponding to the gids allocated to that rank.

For a given timestep, after new activations are calculated and written to the relevant locations in the global activations array on each rank, the arrays are globally synchronized using an in-place MPI_Allgather operation across MPI_COMM_WORLD using a custom strided datatype:

```
MPI_Datatype strided, resizestrided;
MPI_Type_vector(max_count, 1, size, MPI_UINT8_T, &strided);
```

```
MPI_Type_create_resized(strided, 0, sizeof(uint8_t), &resizestrided);
MPI_Type_commit(&resizestrided);
…
void synchronize() {
    MPI_Allgatherv(MPI_IN_PLACE, 1, resizestrided,
                   global_activations, recvcounts, displacements,
                   resizestrided, MPI_COMM_WORLD);
}
```

Where `max_count` is the maximum number of units instantiated on any rank, `size` is the world size, `global_activations` is the activations array, `recvcounts` is an array of 1's with length `size`, and `displacements` is the integer sequence `(0, 1, 2, … size)`.

At the end of the `synchronize` operation, every rank has the up-to-date activation state of every `Unit` in the entire network, even if some of that information will not be used on that rank (e.g., if a layer is frozen).

## Layer-Based Approach

For the layer-based approach, every `Unit` was given a unique index in [0, sizeofLayer) with respect to the `Layer` to which it belonged. We stipulated that only world sizes that are a multiple of four were permitted, meaning that MPI ranks could be distributed perfectly evenly between the 4 model layers. `Units` were then allocated as evenly as possible to ranks assigned to the same layer. As with the round-robin approach, all `Connections` were instantiated on the same rank as the *postsynaptic* `Unit`. Unlike the round-robin approach, the activation arrays were implemented as instance variables of the `Layer` class, and only allocated if either the MPI rank had been assigned to that layer or the layer to which the MPI rank had been assigned had interconnections with that layer (e.g. all layer objects on an HPC rank were allocated activation array memory, however the SC1 layer object was not allocated activation array memory on any rank assigned to layer SC0 as there are not connections between SC0 and SC1). We would expect therefore that overall this approach is more memory efficient. The partitioning scheme is illustrated in Figure 5.

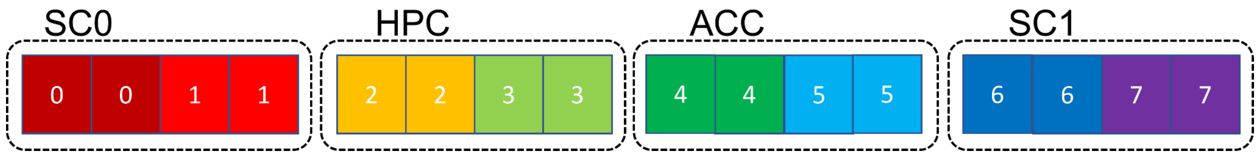


Figure 5. Illustration of the layer-based partitioning scheme for a hypothetical 16 Unit model (4 units per layer) running on 8 ranks. The numbers / colors indicate which rank 'owns' that element of each separate activations array (i.e. is responsible for updating that element prior to synchronization), corresponding to an even (as possible) subset of Unit indices belonging to the

layer to which that rank was assigned. Under this partitioning policy, ranks only instantiate Units from a single layer.

We generated sub intracommunicators for each layer, and during construction of tracts between layers we constructed intercommunicators between the participating layer groups:

```
MPI_Comm layer_comm;
layer_id = world_rank % 4;
MPI_Comm_split(MPI_COMM_WORLD, layer_id, world_rank, &layer_comm);
…
// in void NsSystem::addBiTract()
int layer1_id = layers.at(layer1Id)->intID;
int layer2_id = layers.at(layer2Id)->intID;
MPI_Comm inter_comm;
if (layer_id == layer1_id) {
     MPI_Intercomm_create(layer_comm, 0, MPI_COMM_WORLD, layer2_id,
                          99+(layer1_id + 1)*layer2_id, &inter_comm);
} else if (layer_id == layer2_id) {
     MPI_Intercomm_create(layer_comm, 0, MPI_COMM_WORLD, layer1_id,
                          99+(layer1_id + 1)*layer2_id, &inter_comm);
}
```

All-to-all operations across intercommunicators has symmetric full-duplex behavior such that the contents of each layer's activation arrays can be staged for synchronization across tracts with a single `MPI_Iallgatherv` call. The behavior is illustrated in Figure 6.

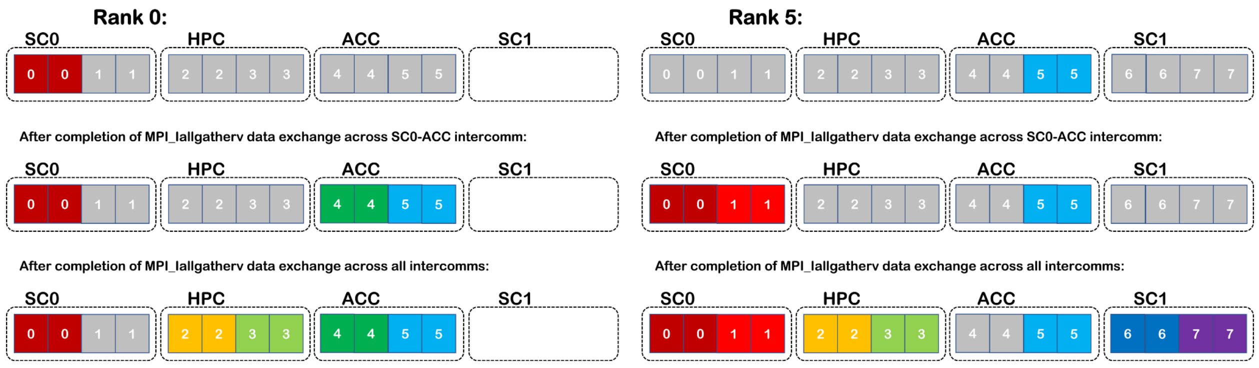

Rank 0:
Rank 5:
SC0
HPC
ACC
SC1
After completion of MPI_Iallgatherv data exchange across SC0-ACC intercomm:
After completion of MPI_Iallgatherv data exchange across all intercomms:

Figure 6. Illustration of collective operations across intercommunicators for state exchange across specific tracts for a hypothetical 16 Unit model (4 units per layer) running on 8 ranks. In this example, rank 0 contains SC0 units and rank 5 contains ACC units. There is no SC1 buffer on rank 0 (or rank 1) as that data is never used by SC0 units. Colored boxes indicate up-to-date activation information. The first row shows the states of the layer-wise activation buffers after local calculation of updated states. The second row shows the states of the layer-wise activation buffers after Allgather across the SC0-ACC intercommunicator: the ACC buffer is fully

populated on rank 0 (and rank 1, not illustrated) and the SC0 is fully populated on rank 5 (and rank 4, not illustrated). The final row shows the final states of all layer-wise activation buffers after all data exchanges for all Iallgather calls have been completed (note: rank 5 / all ranks with HPC / ACC units need to execute 3 MPI collective calls across intercommunicators while rank 0 / all ranks with SC0 / SC1 units need to execute 2 MPI collective calls across intercommunicators, one call per tract between layers).

By communicating on a per-tract basis, we were also able to skip communication between layers that were frozen during a timestep. At the end of synchronization, up-to-date data ends up only where it is going to be used - note that in addition to SC0 and SC1 ranks not allocating buffers for the opposite cortical region altogether, no data is exchanged between any ranks belonging to the same layer during synchronization as there are no intra-layer synaptic connections.

### Deployment Platform and Infrastructure

All simulations and performance tests were run on Knight's Landing (KNL) compute nodes on Stampede2 (Intel Xeon Phi 7250 @ 1.4GHz) running Red Hat Enterprise Linux 7. The Stampede2 network interconnect is a 100Gb/sec Intel Omni-Path (OPA) network with a fat tree topology employing six core switches. The MPI implementation used was Intel MPI.

## Verification

In order to verify the equivalent functionality of our parallel implementations compared to the original model, we first checked that the memory consolidation and reconsolidation behavior was consistent. While there is stochasticity in our model, all three implementations displayed the same trends in recall scores as a function of time of HPC lesion, and became more similar when the data was averaged across multiple trials.

To not only support but prove the equivalency of the implementations, we also ran simulations without stochasticity. While seeding randomness may have been most ideal to prove the equivalency of the implementations, we found it to be difficult to execute and unnecessary. By simply turning off the stochastic  determination of the activation of each unit, so it is purely determined by the weighted inputs and an activation threshold, we removed randomness from our model for a verification test. We ran all three implementations on the same simulation and found the resulting data describing the recall scores across the time of HPC lesion to be exactly the same. This proves our implementations are equivalent and verifies our parallel implementations.

## Performance Evaluation

### Strong Scaling

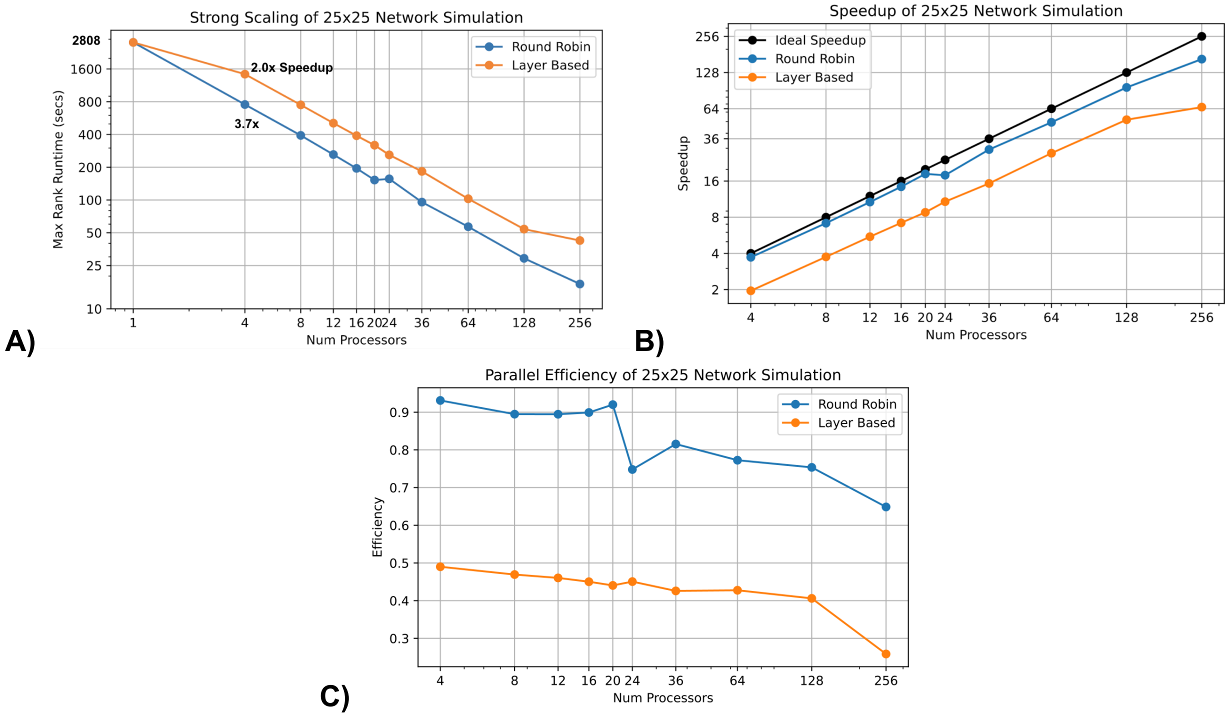


Figure 7. Strong scaling runtime, speedup, and parallel efficiency on a 25x25 unit network and the consolidation lesion simulation with both parallel implementations. A) Max runtime across ranks for a given number of processors. Serial time from the original implementation is plotted at x=1. B) Speedup of parallel simulations of both implementations compared to ideal speedup (black), where the serial time is decreased by a factor of the number of processors used. C) Parallel efficiency across number of processors on the same simulations.

Using both parallel implementations, we performed strong scaling tests to quantify the efficiency of our parallelization. Using the standard network with 25x25 neurons per layer, we ran one simulation of the HPC-lesion task with increasing numbers of processors from 4 to 256. The original serial implementation took 2808 seconds to complete this task. With four processors, the round-robin and layer based implementations took 754 and 1433 seconds respectively to complete this same task, giving them 2.0x and 3.7x speedup respectively. We then see a remarkably linear scaling behavior for both implementations, where the round-robin implementation consistently gets about 80% efficiency and the layer based implementation gets about 45% efficiency. We were able to speed up the original 2808 second simulation to just 17 seconds (165x speedup) using 256 processors.

Although we see strong speedup, the layer based implementation quite surprisingly takes about twice as long as the round-robin implementation across all strong scaling cases. We investigated this behavior using profiling tools to elucidate the cause of this unexpected performance (see Profiling and Discussion sections). Furthermore, the efficiency of the implementations still drops as we increase the number of processors, especially for the largest number of tasks and in the round-robin implementation at 24 tasks.

## Weak Scaling

We performed weak scaling tests to examine the performance of the parallel implementations on not only more processors but also larger problem sizes. We used a linear weak scaling scheme where the number of units in the model was scaled by the same factor as the number of processors. However, since the number of connections scales quadratically with respect to the number of units, the amount of work done by each processor still increased linearly. Thus we expect to see at best linear scaling of the runtime across trials.

The results of our weak scaling tests (Figure 8) show again that the layer-based implementation is about 2x slower than the round-robin implementation across any number of processors. At our largest problem size, a simulation with 1,600 units (320x5) using 256 processors took about 105 seconds. This is still 27x faster than the original serial implementation on only a 25x25 unit simulation.

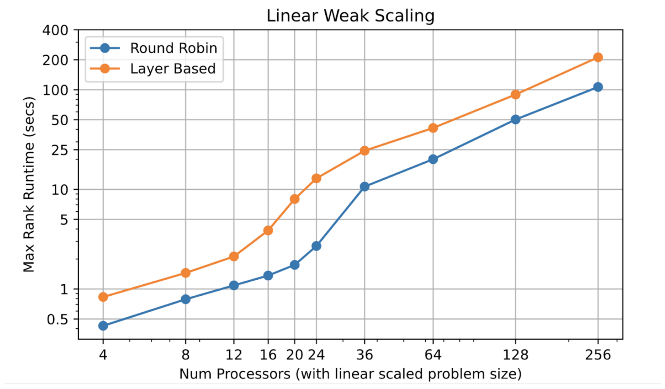


Figure 8. Max runtime of ranks on weak scaling simulations for both parallel implementations. The total number of units in the network are scaled by the same factor as the number of processors, starting from a 5x5 unit simulation with 4 processors, continuing to a 10x5 unit simulation with 8, a 15x5 unit simulation with 12, and so on.

## Memory Scaling

We collected per-task maxRSS data (i.e., maximum amount of memory used by all MPI tasks at any time) using the `time` command for the strong and weak scaling runs as reported above (Figure 9). The round-robin implementation performs better with respect to memory as well, with a consistently lower mean memory usage and lower maximum memory usage by any task.

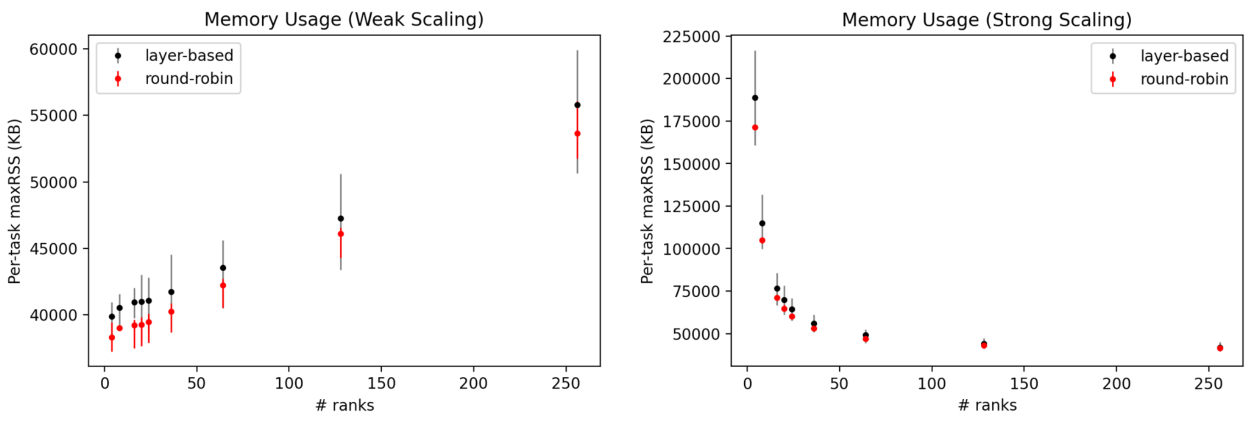
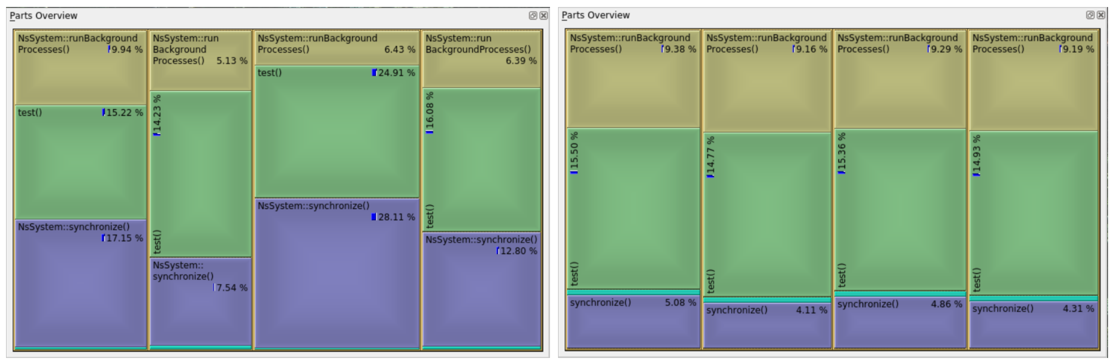


Figure 9. Per-task maxRSS data for weak and strong scaling runs. Dots are the mean max memory usage for all tasks and vertical lines are the range across all tasks.

## Parallel Implementation Profiling

To investigate the performance disparities in the parallel implementations, we profiled their codes using the `valgrind` tool _X_X_X_ (results summarized in Figure X) on a standard lesion simulation using four tasks. For both implementations and all tasks, almost all time is spent computing and updating the new activations (`NsSystem::runBackgroundProcesses` & `test`) and communicating data between ranks (`NsSystem::synchronize`). In the layer based tasks, we see large differences in the proportion of time spent on communication versus computation. The round-robin tasks, in contrast, have extremely consistent proportions of communication versus computation.



Parts Overview
NsSystem::runBackground Processes() 9.94 %
test() 15.22 %
NsSystem::synchronize() 17.15 %
NsSystem::run Background Processes() 5.13 %
14.23 %
test()
NsSystem:: synchronize() 7.54 %
NsSystem::runBackground Processes() 6.43 %
test() 24.91 %
NsSystem::synchronize() 28.11 %
NsSystem::run BackgroundProcesses() 6.39 %
16.08 %
test()
NsSystem::synchronize() 12.80 %
Parts Overview
NsSystem::runBackground Processes() 9.38 %
15.50 %
test()
synchronize() 5.08 %
NsSystem::runBackground Processes() 9.16 %
14.77 %
test()
synchronize() 4.11 %
NsSystem::runBackground Processes() 9.29 %
15.36 %
test()
synchronize() 4.86 %
NsSystem::runBackground Processes() 9.19 %
14.93 %
test()
synchronize() 4.31 %

*Layer based* *Round-Robin*

Figure 10. Summary tables of proportion of runtime spent within functions for four tasks on both parallel implementations of the standard 25x25 network simulation.

## Discussion and Future Directions

Using two parallelization approaches for this network model, we successfully decreased the walltime of large network simulations, achieving a maximum of 165x speedup (17 sec from 2808 sec) with 256 processors on a 25x25 neuron-per-layer model with the round-robin implementation. Furthermore, our parallel implementations have consistent and strong parallel efficiency between 4 and 256 tasks at about 80% and 40% for the round-robin and layer based implementations respectively. From our weak scaling trials, we were also able to drastically increase the number of neurons in the model that can be run in a feasible amount of time (1600 neuron-per-layer simulation in 105 seconds using the round-robin implementation). This demonstrates that our parallelization methodology was very successful in enabling larger network simulations of Helfer and Shultz's model of systems memory consolidation and reconsolidation.

### Examining the performance of the layer based implementation

The layer based parallelization approach, which we thought would be more efficient, showed about half the efficiency and twice the runtime across all tests compared to the simpler round-robin approach. Using our code profiling results, we can examine this behavior and attempt to explain the disparity in performance.

The total run time of a parallel application is determined by the run time of the slowest rank, therefore load balancing is a strong priority. The layer-based parallelization scheme has sub-optimal load balancing in two key respects:

1) Load balancing of communication

We expected the layer based parallel implementation to perform better than the round-robin implementation because it communicates less data between tasks. However, in our strong scaling tests we saw that the layer based implementation achieved only about half of the parallel efficiency as the other. The parallel profiling results can help explain the reasons for this discrepancy.

One of the key differences in time breakdown between ranks in the profiling results are due to imbalances in communication time between ranks. Thus, we see that the layer based implementation has load balancing issues with communication. Considering the implementation, this communication load imbalance is not surprising. All units on a rank must belong to the same region to allow intercommunicator methods. Since the sensory cortex regions do not share connections between them while the HPC and ACC are connected to all other regions, tasks that compute activations for SC0/SC1 units have less data to both receive and send than those for HPC/ACC units, leading to an imbalance in communication load.

2) Load balancing of computation

The graph partitioning policies of both round-robin and layer-based parallelization approaches guarantee at most a discrepancy of one `Unit` with respect to the number of `Units` instantiated per rank. However, this does not fully cover computational load balance as the amount of arithmetic work required to evaluate the new state of a `Unit` varies depending on the number of synaptic connections. Specifically, units in SC0 and SC1 have to compute smaller sums as they have no connections to each other. Additionally, ranks assigned to layers SC0 and SC1 will overall instantiate fewer `Connection` objects for which parameter updates need to be calculated and applied, for example during learning. Therefore, in addition to making overall fewer MPI collective calls, ranks assigned to layers SC0 and SC1 in the layer-based implementation do systematically less arithmetic work than other ranks, further compounding load balancing issues.

3) Suboptimal intergroup communication efficiency

While there is an imbalance of communication and computation load across ranks in the layer based methods, the ranks still have smaller or at worst equivalent communication and computation loads compared to the round-robin ranks. Thus, this load imbalance alone does not explain why the layer based model is so much less efficient.

Our results are consistent with previous work regarding parallel spiking network simulation which has shown that all-to-all exchange is generally superior to MPI point-to-point methods unless the number of ranks is significantly greater than the average number of connections per cell — often on the order of 10000 — since almost every spike needs to be delivered to almost every rank (Lytton et al., 2016). Importantly, algorithms for intergroup communication in production MPI codes are generally not well optimized compared to collective communication across global intracommunicators i.e., MPI_COMM_WORLD (Kang et al., 2019). The work by Kang et al. suggests that algorithmic improvements to the intergroup collective Allgather / Allgatherv operations can yield up to 24x communication speedup over the standard root-gather-broadcast approach adopted by many production codes (e.g., MPICH, MVAPICH, OpenMPI), though we were not able to determine specifics about how intergroup Allgatherv is implemented in Intel MPI. Future work could examine how the scaling efficiency of the layer-based algorithm is affected by implementing these more performant algorithms to achieve intergroup communication instead of using the production versions.

Another instructive insight comes from examining and comparing the memory usage of the two parallelization approaches. The layer-based implementation shows worse memory usage (consistently higher mean and higher maximum across tasks) despite the cumulative model components themselves allocating strictly less than or equal memory compared to the round-robin implementation. The consistently higher maximum is largely explained by the fact that the layer-based implementation clusters units with higher connection counts onto the same ranks (while the round-robin implementation does not). However, there may also be a larger influence of ‘hidden’ memory overhead related to communication, specifically message buffers and connections, which is an MPI implementation (in this case Intel MPI) detail and not directly managed by us or specified by the MPI standard (Goodell et al., 2011). Additionally, there is some memory overhead associated with extra subcommunicators (`layer_comm`) and intercommunicators necessary for the layer-based implementation.

## Potential Improvements

Parallel spiking network models with event-driven communication only stage data for communication if a presynaptic event has occurred. There are no spiking events in this model, however the states that need to be communicated between ranks are not continuous variables, rather they are boolean values 0 or 1. One potential optimization would be to only communicate states that have changed from the previous timestep (the equivalent of a presynaptic ‘event’). Whether or not this would justify the additional overhead of moving data to and from intermediate buffers and dynamically varying the communication pattern over the course of a simulation (instead of, as in our implementations, being able to pre-specify the counts and displacements for all Allgather ops for the entire course of the simulation) will depend on the probability with which the activation of a neuron varies between timesteps.

As noted when describing the parallelization schemes, the class-based abstractions adopted by the original implementation and the resulting data fragmentation are likely not optimal from a performance perspective due to cache inefficiency and the overhead of multiple pointer indirections to access data. A more performant abstraction may be to take a data-oriented approach to design instead of a class-based approach by organizing all the data dependencies (activation states, synaptic parameters) into contiguous arrays and executing the simulation as vector arithmetic over these arrays. This would yield better data locality and cache efficiency. Moreover, an abstraction like that could be ported more easily to GPU to potentially harness massively multithreaded parallelism.

The current network implementation generates some invalid responses when the size of the network gets large, such as recall being worse for the fully intact network than with HPC lesioned. We speculate that this is due to improper normalization of synaptic inputs, and is an issue with how the synapses and neuronal activity are modeled. For this project we didn’t invest time in debugging underlying conceptual issues with the model, however future work may ameliorate these issues by perhaps implementing an adaptive normalization mechanism or inhibitory mechanisms e.g., via inhibitory synaptic connections between neurons within a layer (which is a common observed regulatory feature in biological neuronal networks).

## Other Discussion and Future Directions

While the strong and weak scaling studies showed remarkably consistent scaling behavior across the number of processors used, the overall efficiency did drop gradually with increased processors and we saw a significant dip in efficiency at 24 tasks in the round-robin scheme performance in the strong scaling data (Figure 7). The reason for this dip is unclear, but it may be a result of changes in the function of MPI_Allgather itself, as it may switch to a different implementation after a certain number of tasks is exceeded. The gradual decrease in efficiency for both parallel schemes is due to the baseline stochasticity in the runtime of each rank that causes synchronization delays at each time step. For small numbers of processors, the longer computation time far outweighs these delays, but as we increase the number of processors in our strong scaling trials, the delays become more dominant as the computation time decreases. Furthermore, the more ranks we have the more variance we will have in their runtime, causing longer delays upon synchronization at each time step and a drop in efficiency for both implementations.

Even with these imperfections in the scaling of our parallel implementations, our robust parallelization enables large network simulations that were previously intractable. Future work should focus on taking advantage of these optimizations to add complexity to the model and investigate more complex memory phenomena. It is now possible to add more regions to the memory network to model more complex memory phenomena. For example, adding a representation of the amygdala might be able to add modulation of the strength of memory consolidation based on the perceived emotional power of the stimulus, just as it does in the brain. Larger network simulations may also be able to learn more stimulus pairs on the same network. Future work should focus on investigating the number of stimulus pairs that are able to be learned and maintained with different interventions and network parameters.

This neural network model of systems memory consolidation and reconsolidation is a powerful tool to investigate memory functionality from an abstract level. With our parallelization methodology, we have opened the door to tractibly increasing the size and scope of memory network simulations in order to shed light on more complex phenomena with greater accuracy.

### Code Availability

All code is available at https://gitlab.oit.duke.edu/courses/bme590l_mah148/bme590l_final_project_mah148_ged12. See README.md in the repository for detailed instructions on how to build and run to generate the data used in this report.

## References

Goodell, D., Gropp, W., Zhao, X., & Thakur, R. (2011). Scalable Memory Use in MPI: A Case Study with MPICH2. In Y. Cotronis, A. Danalis, D. S. Nikolopoulos, & J. Dongarra (Eds.), *Recent Advances in the Message Passing Interface* (pp. 140–149). Springer. https://doi.org/10.1007/978-3-642-24449-0_17

Helfer, P., & Shultz, T. R. (2020). A computational model of systems memory consolidation and reconsolidation. *Hippocampus*, *30*(7), 659–677. https://doi.org/10.1002/hipo.23187

Kang, Q., Träff, J. L., Al-Bahrani, R., Agrawal, A., Choudhary, A., & Liao, W. (2019). Scalable Algorithms for MPI Intergroup Allgather and Allgatherv. *Parallel Computing*, *85*, 220–230. https://doi.org/10.1016/j.parco.2019.04.015

Lytton, W. W., Seidenstein, A. H., Dura-Bernal, S., McDougal, R. A., Schürmann, F., & Hines, M. L. (2016). Simulation neurotechnologies for advancing brain research: Parallelizing large networks in NEURON. *Neural Computation*, *28*(10), 2063–2090. https://doi.org/10.1162/NECO_a_00876