NERDSS_Parallel_Developer_Guide

December 2024

Table of Contents

1. Serial NERDSS 1

1.1 Simulation environment 1

1.2 Sub-volume Optimization 2

1.3 Pseudo code for Serial NERDSS 2

2. Parallel NERDSS 5

2.1 Domain Decomposition 5

2.2 Processor Topology 5

2.3 Edge and Ghost Regions 6

3. More details for Data Structures of NERDSS 11

4. More details for Parallel NERDSS Communication 14

5. Debugging 23

1. Serial NERDSS

Serial NERDSS[1] is a particle and rigid-body structure-resolved reaction-diffusion software. The molecule serves as the fundamental simulation object within the system. Each molecule is characterized by a center-of-mass and one or more interfaces capable of binding to a single interface from another molecule. Multiple molecules can form a Complex via interface binding. A single molecule is also assigned to a Complex of size 1. Both intramolecular and intracomplex flexibility are not allowed; all species are rigid bodies.

1.1 Simulation environment

1) A vector of molecules stores all molecules present in the system.

2) Each molecule is assigned a unique index property, corresponding to its position within the vector.

3) Each molecule contains a vector of interfaces.

[1] Varga MJ, Fu Y, Loggia S, Yogurtcu ON, Johnson ME. NERDSS: A Nonequilibrium Simulator for Multibody Self-Assembly at the Cellular Scale. Biophys J. 2020;118(12):3026-3040. doi:10.1016/j.bpj.2020.05.002

4) Each interface is assigned an index property, representing its position within the molecule's interfaces vector.
5) Each interface can have more than one state. And the interface can change its state during the simulation.
6) A vector of complexes stores all complexes formed in the system.
7) Each complex is assigned a unique index property, corresponding to its position within the vector.
8) Each complex has a vector of integers named memberList, storing all indexes of molecules forming this complex.
9) Each molecule has a myComIndex property storing the index of complex to which it belongs.
This organizational structure allows for efficient tracking of molecular interactions and complex formations. An interface can identify its binding partner (both the molecule and the specific interface) by storing the corresponding indexes.

1.2 Sub-volume Optimization

To enhance computational efficiency, we previously implemented a sub-volume division strategy in the serial code. Each sub-volume only evaluates interactions within itself and to adjacent sub-volumes. In 3D, each sub-volume has 26 neighboring sub-volumes (ignoring boundaries).

1) *Cutoff Distance*: We calculate a cutoff distance based on the largest separation between species that can still result in a bimolecular reaction. Beyond this distance, molecules are too far apart to collide and interact with one another.

2) *Spatial Division*: The simulation volume is partitioned into sub-volumes, each with dimensions greater than or equal to the cutoff distance. This division guarantees that binding interactions between interfaces can only occur within the same sub-volume or between adjacent sub-volumes. This significantly reduces the number of interface pairs that need to be checked for potential reactions.

3) *Balance Consideration*: While increasing the number of sub-volumes improves efficiency, an excessive number will slow down the simulation due to the computational overhead of iterating through all sub-volumes and their adjacent neighbors. To maintain the optimal performance in the serial version of the simulator, we do not allow more sub-volumes than there are particles.

1.3 Pseudo code for Serial NERDSS

The following pseudo code describes the reaction-diffusion process in the Serial version of the simulator. Key constraints: Each molecule can participate in only one

reaction per iteration step, even if it has multiple interfaces. It the molecule does not react, it moves as a rigid body. If a complex has multiple molecules, they are each allowed to try and react independently. If a complex does not react, it diffuses as a rigid body in that iteration step. Steric overlaps are prevented.

Reaction-Diffusion Simulation (Serial Version)

```
// Initialization
1. Generate all molecules and corresponding complexes
   Store molecules in vector: moleculeList
   Store complexes in vector: complexList

2. Calculate the cutoff_distance
   Divide simulation volume into sub_volumes based on cutoff_distance
   Store sub_volumes in vector: subVolList
   For each subVol in subVolList:
      Initialize vector<int> memberMolList
   For each molecule in moleculeList:
      Set molecule.mySubVolIndex to appropriate subVol index

// Main Simulation Loop
3. For iteration = 1 to max_interations:
   4. Update subVol memberships:
     For each subVol in subVolList:
        Clear subVol.memberMolList
     For each molecule in moleculeList:
        Update molecule.mySubVolIndex
         Append molecule.index to subVol.memberMolList
   5. Process zeroth-order and first-order reactions:
      Check and perform:
         - Molecule creation
         - Molecule destruction
```

- Unimolecular state changes

- Complex dissociation

6. Calculate and store binding probabilities for all possible second-order bimolecular reactions.

*If the molecule interface already participated in a 0^{th} or 1^{st} order reaction, it can only have binding probabilities set to zero. This applies to all interfaces on the molecule. They must still be evaluated for second-order reactions, so that they will avoid overlap with reaction partners that are close by. The other elements of the complex (if the molecule is part of a complex) are not restricted from reacting, but are restricted from diffusing.

```
For each subVol in subVolList:
  For each molecule1 in subVol.memberMolList:
    For each molecule2 in subVol.memberMolList:
      If binding_possible(molecule1, molecule2):
        Store binding information in molecule1 and molecule2
    For each subVol in adjacentSubVols:
      For each molecule2 in subVol.memberMolList:
        If binding_possible(molecule1, molecule2):
          Store binding information in molecule1 and molecule2
```

7. Perform second-order bimolecular reactions

```
For each molecule in moleculeList:
```

Compare the binding probabilities to a URN. If the probability is >URN, perform the reaction.

*Note, that if this molecule already underwent a binding reaction, via another interface, then its probabilities for binding will all have been set to zero.

Perform bimolecular reactions by associating molecule pair into their defined ‘bound’ orientation.

8. Perform diffusion for unreacted complexes:

```
For each molecule in moleculeList:
```

If molecule has not undergone a 0^{th}, 1^{st}, or 2^{nd} order reaction, or is not part of a complex that has undergone one of these reactions:

Diffuse its complex as rigid body

Ensure no steric overlaps occur with all molecules in its partner list, including molecules that have undergone reactions.

9. Reset reaction information:

For each molecule in moleculeList:

Clear reaction status and information

10. Update simulation time and collect data as needed

// End of main simulation loop

2. Parallel NERDSS

The MPI implementation of the simulator leverages distributed computing techniques to divide the simulation workload across multiple processors and nodes.

2.1 Domain Decomposition

a) The simulation volume is portioned along the x-axis, with each partition assigned to a distinct processor.

b) Each processor is responsible for a subset of the sub-volumes within its assigned region.

c) A processor corresponds to a single MPI rank.

2.2 Processor Topology

a) Processors are arranged in a linear topology along the x-axis.

b) For any given processor n:

- If there is an adjacent processor on its left side, it is referred to as the “left neighbor processor”, n-1.
- Similarly, an adjacent processor on the right side is called the “right neighbor processor”, n+1.

2.3 Edge and Ghost Regions

a) Edge Region: For a processor with a neighbor on one side, the sub-volumes at the boundary along the x-axis are designated as the “Edge region”. The Edge region contains all the molecules that may interact with molecules in the neighboring processor’s domain.

b) Ghost Region: A copy of a neighboring processor’s Edge region. Thus, these sub-volumes are not owned by the processor. Ghost regions ensure the evaluation of reactions that occur across processor boundaries.

The update of Ghost and Edge regions during the simulation is implemented using Message Passing Interface (MPI) functions.

2.4 Extended Properties for Molecules and Complexes

To facilitate efficient parallel processing and inter-processor communication, we introduce additional properties to the Molecule and Complex classes.

a) Global ID: Since each processor maintains its own local indexing system for molecules and complexes, global unique ID ensure unambiguous identification of entities throughout the distributed simulation environment.

b) Spatial Region Flags: Four Boolean properties are added to both Molecule and Complex on each processor:

i. isLeftGhost: True for a complex if any part of it is in the left Ghost region. True for a molecule in the left Ghost region. True for a molecule that is part of a complex in the left Ghost region, and that molecule is not in the left Edge region.

ii. isLeftEdge: True for a complex if any part of it is in the left Edge region. True for a molecule in the left Edge region. True for a molecule that is part of a complex in the left Edge region, and that molecule is not in the left Ghost region.

iii. isRightEdge: Same conditions as above.

iv. isRightGhost: Same conditions as above.

c) Region Assignment Consequences:

i. Molecules can only be ‘True’ for one of these four regions, or ‘False’ for all of them.

ii. Complexes can be ‘True’ for two regions if they span both an Edge and Ghost region.

iii. Every single molecule in a Complex that has a 'True' flag will either be assigned as a Ghost or Edge molecule. This includes molecules that extend out of both the ghost and edge sub-volumes. These molecules must be assigned a sub-volume on the physical processor, even though molecules beyond the ghost region exist outside of it. They are assigned to the closest sub-volume by retaining the y and z index of the sub-volume, and setting the x index to 0.

d) Communication Tracking: A Boolean property receivedFromNeighbor is added to both Molecule and Complex classes. This property is used to track the loss of molecules and complexes from the observed processor during the inter-processor communication.

e) Communication Protocol:

a. Before communication: receivedFromNeighbor is set to false for all molecules and complexes in Edge and Ghost regions.

b. After receiving data from a neighbor processor: receivedFromNeighbor is set to true for all received entities.

c. Post-communication cleanup: Any molecule or complex in Edge or Ghost regions with receivedFromNeighbor == false is considered not received back from the neighbor and is deleted from the current processor.

2.5 Pseudo-code for Parallel NERDSS

```
1. Initialization:
  1.1 Initialize simulation domain and parameters
  1.2 Partition simulation volume along x-axis
  1.3 Assign partitions to available processors


2. Pre-Process Setup:
  For each processor:
   2.1 Initialize local sub-volumes, moleculeList, and complexList
   2.2 Assign global IDs to molecules and complexes
   2.3 Identify Edge regions based on the sub-volume indices. If the
sub-volume x-index is 1, it is a left edge, if the sub-volume x-index
is N-2, it is a right edge. The x-index is 0 for the left ghost-
region. The x-index is N-1 for the right ghost-region.
```

2.4 Set spatial region flags for molecule and complex, by looping over all molecules in the edge regions, and including all molecules that are shared by their parent complexes.

3. Main Simulation Loop:

For each time step:

3.2 Process Local Reactions:

3.2.1 Check and perform zeroth-order and first-order reactions for all molecules in your physical processor, **excluding the right edge region**. Check and perform these reactions for your left ghost region, and **exclude the right ghost region.**

3.2.2 Check for potential second-order bimolecular reactions for all particles in your physical processor. Include both the left edge and the right edge region. Include both the left ghost region and the right ghost region.

3.3 Divide Processor region:

Split your processor's physical region into Left half and Right half from middle of x-axis. Create two lists, a left half list that contains the index of all molecules in the left half. A right half list contains the index of all molecules in the right half.

3.4 Process Left half:

3.4.1 Perform second-order bimolecular reactions for molecules in your physical left half and include your left ghost region. This includes all molecules that are assigned to the ghost region, even if it is due to being a member of a complex, and its physical position is outside of the ghost region. If a molecule associates, it has a new flag set to 'isAssociated=true'.

3.4.2 Diffuse unreacted complexes (excluding left ghost region). If a molecule is part of a complex that is partially in the ghost region, it is not allowed to diffuse. Don't diffuse complexes that spread across the left/right half and have not reacted. That will cause molecules to diffuse on the right half, even though they have not yet tried to react. Molecules can also diffuse into the right half, from the left half. Molecules can also diffuse into the left ghost region, from the left edge region.

3.4.3 Update sub-volume memberships and spatial region flags for all the molecules in your processor, including the ghost regions. The right ghost region should not have changed. This includes ghost molecules that are physically outside of your sub-volumes, but are part of a ghost-region complex.

3.5 Left-to-Right Communication:

If not last processor:

Set the receivedFromNeighbor to false for right edge and right ghost regions

If not first processor:

Send the left edge and ghost data to left neighbor processor

If not last processor:

Receive the data from right neighbor processor

Update the right ghost region and right edge region. That means that all molecules and complexes in these two regions either already had 0^{th}, 1^{st}, or 2^{nd} order reaction performed, or, they attempted but did not perform any reactions. Those reaction probabilities should all have been set to zero. Some of these molecules (most of those in the right ghost region) will already have diffused.

Delete unreceived molecules and complexes in right ghost/edge regions. This should only include molecules that were in your right ghost region, as they may have diffused beyond further to the right. It should not include any molecules in the right edge region, which could maximally have moved into the right ghost region, or if they diffused left you had assigned them to sub-volume 0 on the neighbor's processor.

3.6 Process Right half:

3.6.1 Perform second-order bimolecular reactions for all molecules in the right half based on the list 'RightHalfIndexes' established in 3.3. This excludes all molecules that diffused into the right half. This should exclude reactions to all molecules that were in the left half, as those reactions were already attempted.

This loop excludes all molecules in the right edge and ghost regions. However, molecules can attempt to react with any molecule in the right edge or right ghost region. These reactions are rejected if

those molecules already underwent a 0th, 1st or 2nd order reaction. The isDissociated and isAssociated flags are used for this.

3.6.2 Diffuse unreacted complexes. All molecules that are in the right half, including molecules that are part of a complex that extends into the left half. All molecules that are in the right edge region that already diffused (via trajStatus::propagated = true flag) are not allowed to diffuse again. All molecules that are in the right ghost region and do not extend into the right edge region would have already diffused. Molecules that are in a complex that spans the edge and ghost might need to diffuse.

3.6.3 Update sub-volume memberships and spatial region flags

3.7 Right-to-Left Communication:

If not first processor:

Set the receivedFromNeighbor to false for left edge region and left ghost region

If not last processor:

Send the right edge and ghost data to right neighbor processor

If not first processor:

Receive the data from left neighbor processor

Update the left ghost region and left edge region

Delete unreceived molecules and complexes in left ghost/edge regions

3.8 Post-Processing:

3.8.1 Reset reaction information for all molecules

3.8.2 Update sub-volume memberships, update spatial region flags for all molecules and complexes

3.8.3 Update simulation time and collect data as needed

4. Finalization:

4.1 Merge results from all processors

4.2 Generate final output

3. More details for Data Structures of NERDSS

3.1 Coord

Used for storing 3-D coordinates. Each coordinate (x, y, and z) is represented as a double.

3.2 SimulVolume

The source of simulation volume information. The SubVolume structure, contained within the SimulVolume structure contains information about the "cells" of the partitioned volume:

xIndex, yIndex and zIndex, cell dimensional indices ({x,y,z} tuple)

absIndex, flattened dimensional index { = xIndex + yIndexNx + zIndexNx*Ny };

memberMolList, a list of indices of molecules within a cell.

neighborList, a list of neighbor cells indices (neighbor sublist supports unique pairing).

> A list of these cell structures is maintained in the SubVolume structure as the subCellList list (vector<SubVolume> subCellList).

Another structure defined in SimulVolume is the Dimensions structure. Its x, y, and y int members contain the number of cells (subvolumed) in each direction, and tot, the total number of sub-volumes.

3.3 MolTemplate

The properties of each type of molecule are contained in a molecule template structure, MolTemplate, and each molecule has an index (molTypeIndex) that references its molecule type.

The MolTemplate structure also contains interfaceList, a list of Interface structures that have information about the type's interfaces for binding. This Interface is different from another interface structure, Iface, that EACH molecules contains. (The molecule's Iface structure contains mutable information: coordinates, state, etc.) The Interface structure contains information of a MolTemplate's interfaces. Non-mutable state information for each interface is contained in a State structure. (e.g. Iface participates in: myForwardRxns,myCreateDestructRxns, rxnPartners or stateChangeRxns.) If an interface is associated with a state, then a reaction involving that interface will depend upon the

state. (e.g. phosphorylated proteins participates in a different set of binding events than unphosphorylated proteins).

A molecule template is often referenced for properties of a molecule and its interfaces. An important property is the monomerList which lists molecule indices for monomers.

3.4 Reactions

Reactions (forward, backward, and create/destruct) determine which interface can participate in bi-molecular association/dissociation and unimolecular create/destroy activity, possibly depending on the state (of an interface).

Reactions can have a forward only direction (->), or forward and backward directions (<->) The types of reactions are:

bimolecular Rxn [association/dissociation]

biMolStateChange [X(state1) + Y -> X(state2) + Y]

uniMolStateChange [unimolecular state change reaction (X <-> X*)]

zerothOrderCreation [creation reaction from concentration (0 -> X)]

destruction [destroys entire molecule/complex, not just interface]

uniMolCreation [creation reaction from Molecule (X -> X + Y)]

3.5 Molecule

Structure Molecule is the most important structure in NERDSS. It is the container for molecule AND interface information. A list of all molecules (structures) is maintained in main as vector<Molecule> moleculeList. A molecule is identified by its index (position) in this vector. This index is stored as a member in the Molecule structure.

Some important fields are:

index - vector position of this molecule in moleculeList

partnerIndex - bound partner index in molecule list

partnerIfaceIndex - interface index of the Molecule's partner.

interaction - a structure holding data related to the interaction, including partner index, and interface index.

comCoord - center of mass coordinate of a molecule

isEmpty - if true, the molecule has been destroyed and is void

numberOfMolecules - counter for the number of molecules in the system. This field is static, i.e. there is only one value for all molecules.

emptyMolList - list of indices to empty Molecules in moleculeList

Iface

Molecule also contains the interfaceList vector of Iface structures. These interfaces contain mutable information such as absolute interface coordinates (coord), the current chemical state (stateIndex), boundness (isbound), etc. and bonding information to other molecules with which it forms a (multi-molecule) complex.

Interaction

It is the Interaction structure in each Iface member that contains the molecule index for the “interaction” partner of the interface.

partnerIndex - bound partner molecule index

parnerIfaceIndex - interface index of partner

conjBackRxn - back reaction for the interaction

3.6 Complex

Initially (in the beginning of a simulation), every molecule is a complex. Hence, initially every molecule has its own Complex structure (coincidentally and initially the index for the molecule and complex are the same). Just like moleculeList, a list of all Complex (structures) is maintained in main as vector<Complex> complexList. The myComIndex variable in the molecule structure is the index (position) in complexList of its associated complex. This index is also stored in an Index variable of the Complex structure.

When a bond forms, one of the complex structures of the molecule becomes the complex (the other complex is marked isEmpty).

The following fields are important:

index - index of this complex in complexList.

memberList - list of complex member molecules (indices).

isEmpty - true if the complex is not in use any more (is a void, waiting to be destroyed)

numberOfComplexes - total number of complexes in the system. This field is static, i.e. there is only one value for all complexes.

comCoord - complex’s center of mass coordinate.

4. More details for Parallel NERDSS Communication

4.1 Messaging Functions

Distributing the initial data among ranks and updating between ranks is performed by MPI messaging. Packing and unpacking the data for communication is performed by serialization and deserialization routines, respectively. Due to the large number of disparate and nested structures, the use of MPI derived types (containers with descriptive formatting) was considered an unnecessary complication. Serialization of an object converts the data that the object holds into a single array of raw bytes, so that it can be transferred over the network in a single MPI transfer (or placed in a binary file with a single write statement for checkpointing). It also allows for picking the fields that need to be transferred, avoiding non-mutable ones. Deserialization is the opposite process. It restores objects from the raw data received through the network (or from a file for checkpointing). Since serialization and deserialization is a "bitwise" copy procedure, structures with substructures must be mined recursive. That is, objects contained in an object are serialized (and deserialized) separately. APIs functions exist for each type of structure, and includes templated forms for list, maps, etc. This "component" approach makes it easy to add a new structure and lists, and their APIs, for the de/serialization process. Base types are serialized and deserialized using macros that need no modification.

The serialized (packed) data are stored in an array of bytes, arrayRank, and sent as a single object to another rank. (The suffix Rank in variable names denotes it is for another rank). The following code illustrates how a primitive variable type, here the double x, is stored in arrayRank, starting from byte zero.

The storage address of arrayRank is determined by &(arrayRank[0]). This pointer is cast into the pointer for type double by double * and finally the value of x is stored in the first eight bytes of arrayRank by *(...) = x.

```
*( (double *) &(arrayRank[0]) ) = x;
```

The next object is stored in arrayRank at the next empty byte. In this case, the next free position in the array is after the double, or sizeof(double) bytes relative to the base address:

```
*( (double *) &(arrayRank[sizeof(double)]) ) = y;
```

More practically, the next free location is maintained in an integer nArrayRank variable, which is updated after each new object is serialized, as shown here for y:

```
*( (double *) &(arrayRank[nArrayRank]) ) = y;
```

nArrayRank += sizeof(double);

At the end of the serialization process, the size of the array arrayRank that needs to be sent to another rank is nArrayRank. (Often the first entry is size information.)

Deserialization is the opposite of serialization. The location of the value for variable y is arrayRank[nArrayRank], where nArrayRank is updated to the next un-deserialized storage location right after. To extract the correct size and type from that position for the assignment into y, the address at a position is determined by &(arrayRank[nArrayRank]), and then the address is cast into a double pointer, (double *). Finally, this “pointer” (lvalue) is dereferenced, *(…) as shown below for y. This deserialization method is shown in the following source code:

y = *((double *) &(arrayRank[nArrayRank]));

nArrayRank += sizeof(double);

To summarize the above examples, variables of type double are serialized to, and deserialized from, an array. Initially, nArrayRank is set to 0. After all the objects have been serialized or deserialized, nArrayRank contains the total occupied storage size.

The process can be easily implemented for any base type using a template. Building on the base variable serialization presented above, functions for serializing STL sequential containers (vectors, lists, maps, matrices, etc.) and structures are created with a generic type declaration:

template <typename T>

where a function argument is typed generically with the template parameter, T (generic type or template parameter) for function arguments and variables, as shown here for typeing a vector:

std::vector<T> to_serialize;

This enables a single definition of functions for various lists of base types. For example, the template function serialize_primitive_vector is declared as:

template <typename T>

void serialize_primitive_vector(std::vector<T> to_serialize, unsigned char *arrayRank, int &nArrayRank);

where arrayRank and nArrayRank are storage variable arguments required for serialization, as already explained.

The function call for a vector of type int containing integer molecular indices (emptyMolList) becomes:

serialize_primitive_vector<int>(emptyMolList, arrayRank, nArrayRank);

Analogously, the template function deserialize_primitive_vector is declared as:

template <typename T>

void deserialize_primitive_vector(std::vector<T> &to_deserialize, unsigned char *arrayRank, int &nArrayRank);

and the symmetrically looking deserialization function call is also easy to write and read:

deserialize_primitive_vector<int>(emptyMolList, arrayRank, nArrayRank);

(Compared to the typical function call, this function template call has an <int> added between the function name and the argument list which instructs the compiler to create (instantiate) a function based on the template for a given type (int).

Developers who introduce a new member in a class/structure that needs to be exchanged between ranks, should insert a serialization and a deserialize function call (of appropriate type) for the member if its class/structure includes serialization/deserialization methods for exchanging data.

Functions for serializing and deserializing a matrix container are:

template <typename T>

void serialize_primitive_matrix(std::vector< std::vector<T> > to_serialize, unsigned char *arrayRank, int &nArrayRank);

and

template <typename T>

void deserialize_primitive_matrix(std::vector< std::vector<T> > &to_deserialize, unsigned char *arrayRank, int &nArrayRank);

Containers that require a size type, can be accommodated by a slight modification to the above functions, as for this vector-of-arrays serialization:

template <typename T, std::size_t S>

void serialize_vector_array(std::vector< std::array<int, S> > to_serialize, unsigned char *arrayRank, int &nArrayRank);

where S is substituted with an constant int, as in:

serialize_vector_array<int, 3>(crossrxn, arrayRank, nArrayRank);

Matrices are serialized and deserialized using the following template functions:

template <typename T>

void serialize_abstract_matrix(std::vector< std::vector<T> > to_serialize, unsigned char *arrayRank, int &nArrayRank);

and

template <typename T>

void deserialize_abstract_matrix(std::vector< std::vector<T> > &to_deserialize, unsigned char *arrayRank, int &nArrayRank);

If one needs a custom serialization function, the implemented template functions should be examined first. If a new function needs to be implemented, please follow the naming conventions of the implemented template functions.

4.2 Serializing and Deserializing Macros

In the previous section, the serializing/deserializing functions were described by their basic operation: packing data into an array of bytes, arrayRank, and recording the total number of occupied bytes of the array in nArrayRank. We illustrate the serialization operation, again, for an integer (x):

*((int *) &(arrayRank[nArrayRank])) = x;

nArrayRank += sizeof(int);

The complicated syntax of the first statement hides the simplicity of the bit-wise copy operation and even masks what is happening if one is not familiar with casting variables into different types. Macros are introduced to regain simplicity and readability of the code. The simple, single-argument PUSH(variable) macro is used to specify the serialization of a primitive type (variable) to serialize: The macro is general in that it works with all primitive types.

The two serialization statements discussed above (the copy variable into the arrayRank array of chars, starting from nArrayRank bytes, followed by increasing nArrayRank by the number of serialized bytes), can be replaced by a macro call. The macro uses (__typeof__) for the compiler to discover the type and size of the argument.

The following code is the PUSH(variable) macro:

```
#define PUSH(variable) \

*( (__typeof__ (variable) *) (arrayRank + nArrayRank) ) = variable; \

nArrayRank += sizeof(variable);
```

where arrayRank and nArrayRank are assumed to be in scope, and the type and size of the primitive type argument are extracted from the variable itself (variable) with the __type__ and sizeof operators.

Similarly, deserializing from the arrayRank array of chars into the variable variable for any of the primitive types starts from the nArrayRank-th byte of arrayRank. It is followed with an increase of nArrayRank by the size of variable variable in bytes.

The POP(variable) macro is:

```
#define POP(variable) \

toSerialize = *( (__typeof__ (variable) *) (arrayRank + nArrayRank) ); \

nArrayRank += sizeof(variable);
```

The following code uses PUSH and POP to serialize and deseriale primitive type variables (and expressions):

```
int a;

PUSH(a);

...

int b;

POP(b);

// and for an expression

PUSH(15+4);
```

4.3 Auxiliary Structures for Parallel Execution

The MpiContext structure is a container for most of the data related to parallel execution. This is the only parallel structure passed “around” as a single, end argument to many functions.

A single parallel container object is used (instead of various individual structures) because the serial implementation of the simulator has deep call stacks (many-level nested function calls) as large as 10, and propagating various parallelization structures down

through the chain to lower level functions would require substantial modification to argument lists. (It would be error prone and time consuming.)

The single argument (container) approach obviates argument changes in future parallelization modifications. It is also easy for a developer to see which functions are possibly modified to support parallel execution when anmpiContext argument is seen in the source function call or in the call stack when debugging. Note that the MpiContext is unique for a rank, which enables it to be defined as a global variable, so that no modifications to serial code function declarations are needed, but it is best to avoid defining variables as global.

The following code snippets show and explain the data members of MpiContext, and function definitions. Unimportant ones are omitted, and the data members are presented in a logical order rather than the order found in the code.

MpiContext is defined as the typedef for the structMpiContext structure, and the former is used throughout the code in the instantiation and reference to the structure:

typedef struct structMpiContext{ // holds MPI related data

...

} MpiContext;

The rank number and size are stored in rank and nprocs. These are used to determine neighboring ranks for shared zones and identifying ranks involved in shared complexes:

int nprocs; // number of MPI processes

int rank; // MPI rank number

The current simulation iteration number is also stored in mpiContext, so that it can be used in any function having a mpiContext pointer. This is useful when debugging with print statements, since it allows printing to begin after a certain number of iterations. The iteration number is:

int simItr;

As explained above, the parallel adaptation of functions uses the mpiContext structure to contain access to serial structures that were hithertofore not needed in the function. This is accomplished by creating pointers to structures, membraneObject, simulVolume, moleculeList, and complexList in main that may be needed for parallel processing or debugging:

Membrane *membraneObject;

SimulVolume *simulVolume;

std::vector<Molecule> *moleculeList;

std::vector<Complex> *complexList;

The following fields hold pointers to byte array storage (MPIArray buffers) for To/From Send/Recv operations to Left/Right ranks, array position integers, and storage size. The MPIArrays storage is assigned by malloc and if occupation approaches capacity the size is increased by 20 percent. They are specified as:

unsigned char* MPIArrayToRight;

unsigned char* MPIArrayFromRight;

unsigned char* MPIArrayToLeft;

unsigned char* MPIArrayFromLeft;

int nMPIArrayToRight;

int nMPIArrayFromRight;

int nMPIArrayToLeft;

int nMPIArrayFromLeft;

int sendBufferSize, recvBufferSize;

The following lines define Send/Recv Request identifiers (MPI_Request type) and status objects (MPI_Status type) for non-blocking communications. These are used to wait for outstanding Send/Recv communications:

// non-blocking identifiers for synchronization

(waiting)

MPI_Request requestSendToLeft, requestSendToRight, requestRecvFromLeft,

requestRecvFromRight;

MPI_Status statusRecvFromLeft, statusRecvFromRight; // contains byte count, etc.

Each rank holds a binning offset, xOffset, so that it can determine its local x-bin. The get_x_bin() function performed this calculation:

int xOffset;

inline int get_x_bin(MpiContext &mpiContext, Molecule &mol){

return int(

(mol.comCoord.x + (*(mpiContext.membraneObject)).waterBox.x / 2) /

(*(mpiContext.simulVolume)).subCellSize.x)

- mpiContext.xOffset;

}

where the molecule's x coordinate (adjusted to a scale of 0 to Xrange from -1/2 Xrange to +1/2 Xrange) is divided by the size of a zone (cell) to get a global number in the x dimension, and then the xOffset offset is applied such that the local numbers begin at 0. Note the use of (*(mpiContext.simulVolume)). It dereferences a pointer (contained in mpiContext) to the simulVolume structure. As explained above, parallel adaptation of functions uses the mpiContext structure to contain/access structures that were not needed in functions before parallelization. The following comments address the case when bin distributions are not even across ranks.

// When nprocs does not divide total_no_bins evenly, the remainder is

// distributed by adding a single cell to each rank, beginning with

// rank 0, until the remainder count is exhausted.

// e.g. distribution for dividing 12 bins onto 5 ranks: 3, 3, 2, 2, 2

For this case, the xOffsets for ranks {0,1,2,3,4} are {0,3,6,8,10}.

Fields startCell and endCell denote the x-bin of the first and last owned (shared) zones, and fields startGhosted and endGhosted x-bins of all zones that the rank can see (i.e, ghost zones are included):

int startCell, endCell;

int startGhosted, endGhosted;

MPI Buffer Space

There are still "magic number" constants used for the communication buffers (an adequate number of bytes needed, for a wide range of transfers, to handle an expected maximum number of outstanding non-blocking Send/Recv operations). Even though the sizes of these arrays for communication between ranks are self-growing, they are set reasonably high for the worst-case scenario at the present scale for the first iteration. For different realms of scaling the initial values may fail for the first iteration.

4.4 Preparing Data Structures for Parallel Execution

A significant amount of coding is dedicated to ingesting the user input, and it uses an insignificant amount of total execution time. To preserve as much of the serial coding as possible, only rank 0 parses the input (as if it is a serial execution).

So, rank 0 has the whole-run (called “serial-specific”) data (molecules, complexes, cells, etc. structures) in lists such as moleculeList, complexList, subCellList, etc. Rank 0 then partitions data (molecules, complexes, cells etc.) into a subset of the data for performing work on a particular rank (called “rank-specific” data). Also, these data structures now contain a few new fields that pertain only to the parallel implementation, call “parallel-specific” fields, which can be disregarded for serial development and execution.

Initially (for serial and parallel execution), a Molecule structure exists for every molecule, and it has a unique Molecule.index that refers to its position in the vector moleculeList of Molecule objects. These index values range from 0 to Nall-1 (Nall=total number of simulation molecules). For parallel execution, each rank will have a moleculeList with a subset of molecules having a index range of 0 to Nrank-1 (Nrank = number of molecules in rank’s subset, sum(Nranks)=Nall). Since the serial-specific index numbers are unique, they are assigned to the rank-specific molecules as Molecule.id. Hence for parallel execution, the molecule index is a local number (used for looping in the serial functions), and the id is a global number (used for uniquely identifying molecules when they cross ranks). For convenience, map arrays (mapSerialToParallelMolecule and mapParallelToSerialMolecule), are created to map between the rank-specific index and the id. (These two maps, as well as serial-specific moleculeList only exist up to and during data partitioning.)

The basic partition operation is to select molecules, complexes, and cells out of the serial-specific moleculeList, complexList, and subCellList lists, and populate the rank-specific moleculeListRank, complexListRank, and subCellListRank lists with molecules, complexes and cells for a rank: rank-specific molecules for a given rank are inserted into a rank-specific moleculeListRank list, rank-specific complexes for a given rank are inserted into a rank-specific complexListRank list, and rank-specific cells for a given rank are inserted into the rank-specific subCellList list of rank-specificSimulVolumeRank.

Generally, as remarked earlier, the rank-specific lists have their usual names with the Rank suffix. (Maps for converting between rank-specific and serial-specific indices are prefixed with a map prefix.)

In the process of selecting data structures for a rank, copies of data structures (Molecule, Complex, simulVolume, etc.) are created (rank-specific structures) and the selected

serial-specific structure content copied into them. Appropriate indexing is modified, parallel-specific fields are assigned values, and the structures are added into new rank-specific lists:

Simulation Volume and Cells

The following are important simulation volume definitions, gathered and stated here as a quick reference: simulation volume - the cubic box in which the simulation occurs. Coordinates range from -WB.x to WB.x/2, -WB.y/2 to WB.y/2, and -WB.z/2 to WB.z/2 (units=nm), where {WaterBox.x,WaterBox.y,WaterBox.z} triple are the volume dimensions (are idential, abbr.=WB).subvolume - subdivisions of the simulation volume. These subdivisions consist of N divisions in each dimension (Nx, Ny and Nz, all equal). cell - a subvolume (alternate reference for a subvolume, and even variables are named as "subCell"s in the serial implementation). subCellSize - {subCellSize.x, subCellSize.y, subCellSize.z} triple of cell x, y and z sizes (identical).

For N ranks there are N partitions as explained in the Data Partitioning section above.

When the input is parsed, each molecule is assigned to a region (cell in SimulVolume.subCellList[]) in the simulation volume. Also, for parallel execution the cells have been partitioned into a set of N contiguous (x-bins) along the x axis, as explained in Data Partitioning above. These partitions are created in the init_x_domain_and_offset() function, and the ranges set as startCell, endCell, startGhosted, endGhosts in the general MPI container, mpiContext. A feature of partitioning in the X-direction is that the necessary data update exchanges are simply implemented as point-to-point nearest neighbor MPI communications.

Data preparation for the ranks exploits the molecule aggregation in cells. For each rank, a loop iterates over all cells. If the cell is within the x-bin's of the rank, between startCell and endCell and including ghosted cells (between startGhost and endGhost), an inner loop iterates over all molecules of the cells memberlist, and applies the basic operations on the partitioned data.

5. Debugging

Relative to a serial code, parallel code is harder to debug due, to the concurrency in execution and handling multiple output streams. The gdb utility can be used with mpirun by starting each MPI process (rank) with a separate gdb instance. In this manner, each rank is assigned a separate terminal window.

Debugging the code often requires tracking a single molecule over ranks during certain iterations. For this purpose, function debug_print, as well as the DEBUG_MOL and DEBUG_FIND_MOL macros were implemented.
Macros only call debug_print, as follows:

```
#define DEBUG_MOL(s) { debug_print(mpiContext, mol, s); }
#define DEBUG_FIND_MOL(s) { for(auto &mol:moleculeList) debug_print(mpiContext, mol, s); }
```

Function debug_print can be modified to print what is of interest for debugging purposes.
Macro DEBUG_MOL
should be called from a function where a molecule is known. If a molecule from moleculeList is to be examined,
use macro DEBUG_FIND_MOL to access it.
Various functions with the “debug_” suffix have also been implemented for debugging purposes:

- debug_firstEmptyIndex
- debug_bndpartner_interface
- debug_molecule_complex_missmatch.

These functions can be found in the src/debug directory of the project, and are useful for early detection of potential problems.
While developing a simulator, especially parts that affect parallel execution, a programmer is strongly encouraged to use and modify these functions as well.
Sometimes, a coding error might be introduced that is not readily detectable. For this purpose, it is strongly suggested that at least once before pushing the code to the repository the programmer call "debug_*" functions in each iteration as a means to check for “impossible-situations” errors that propagate into these data structures.