**A summary of where AMSI is and exactly what things are needed to start solving some specific multiscale tissue problems.**

The Adaptive Multimodel Simulation Infrastructure (AMSI) has been developed to provide a set of support libraries and functionalities for the development, implementation, and execution of general multimodel simulations. Key areas of functionality for AMSI include managing the parallel execution space of multimodel simulations, scale-linking communications, and dynamic scale-sensitive load-balancing operations and adaptive procedures.

AMSI provides runtime support for these key areas through maintaining a real-time minimal meta-data model of various qualities of the simulation, with respect to parallel execution requirements for simulation scale-tasks, scale-linking operations and the actual data representations of such scale-linking communications, and many additional quantities. This meta-data itself is intended to be generalized and extensible such that additional functionalities depending on an extended meta-model may be implemented by AMSI users, though at present the meta-data system is limited to managing those pieces of meta-data required for currently implemented AMSI functionality.

The current implementation of AMSI includes support for general static discretization of the parallel space into scale-tasks, which provides an environment particularly suited to the implementation of hierarchical multiscale simulations, though not explicitly inhibiting the development of concurrently parallel simulations. (Additional discussion of parallel space discretization).

Using the data structures describing the static discretization of the parallel execution space - created at initialization time - the multiscale simulation may dynamically (or statically) determine which scales will need to interact. This approach is taken in order to reduce unneeded overhead caused by maintaining data structures specific to interacting scale-tasks, which would be the case when assuming that any two scale-tasks may interact.

Scale-linking quantities are dynamically registered with the AMSI system by the scale-task which produces the data, along with information about which interacting scale-tasks will receive how much data. This data is stored in AMSI, and when scale-linking communication is to take place, a user-configurable algorithm operating on a mapping of all pertinent scale-linking data is used to provide a full plan of scale-linking communication. This plan is distributed (minimally) to those processes which need to know their individual role in the communication. At this point scale-linking communication of any POD datatypes may take place at any time. If the quantity of scale-linking data changes at any point the previous steps must be repeated, but if the quantity remains the same for several instances of inter-scale communication the processes do not need to occur every time.

Multiscale load balancing processes also make use of the meta-data structures concerning the static discretization of the parallel space. AMSI currently has separate functionalities for: data removal, data addition with load balancing, and migration. Each functionality currently works within a single communication pattern between two tasks. The removal routine does not automatically call any load balancing or migration routines and should normally be paired with the migration routine. In the current hierarchical mutliscale test problem these routines are called in the previously listed order. Data is first removed which may unbalance the work loads. Next, data is added with load balancing so that differences in workload can be filled by the new data. Finally, migration is called to rectify any remaining unbalance. Currently each of these routines also assumes implicit ordering of data within AMSI since explicit ordering has not yet been implemented. This means that when using the load balancing and migration routines, the user's understanding of the data outside of AMSI needs to be reordered to maintain agreement across tasks. Specifics regarding each of these processes are provided at the end of the document.

These processes may operate by giving deference to higher scales in a hierarchical multiscale setup, or by treating all scales as equivalent. For instance, in the AMSI-supported implementation of the soft tissue mechanics multiscale simulation (Biotissue), single-process tertiary scale RVEs are grouped together in the same tertiary scale-task, wherein each process receives a buffer of RVEs to be computed locally (minimally a single RVE but typically many more). Changes to the mesh at the engineering scale - in which the tertiary-scale RVEs are located at numerical integration points - result in differing total numbers of RVEs, and possibly in the implicit ordering of the RVEs to change. While the impetus for the load-balancing operation is multiscale, the load-balancing operation to create/destroy and redistribute RVEs in the tertiary scales parallel execution space partition only operates on the tertiary scale itself -- treating the ‘higher’ engineering scale as fixed. Fully multiscale load balancing operations will work on the premise that data redistribution may take place on either or both scales simultaneously, though this would predominantly be to take advantage of data locality for interscale communication neighborhoods, and represents an area of future work.

Discussion of tasks related to NIH grant:

Specific Aim 1. We will continue to develop our multicomponent model of quasistatic prefailure behavior, incorporating cells and interfibrillar material  
  
Task 1.1. We will construct a three-scale model including the tissue, the cell, and the network scales.   
Task 1.2. We will incorporate interfibrillar material into the model both as a parallel component (rule of   
mixtures) and by direct finite element modeling.   
Task 1.3. We will develop a selective / adaptive modeling framework to reduce the computational demands of   
the model and make it tractable for realistic geometries.   
Task 1.4. We will perform a series of experiments to specify and test the models  
  
  
Task 1.1 is approaching 2/3 complete. Nothing in AMSI itself is preventing the accomplishment of this entire task, we simply need to complete writing the fiber-matrix scale and completely write the cellular scale. Given our approach to writing the tissue and fiber-matrix network scales as modular and general as possible, introducing a cellular scale should take fewer man-hours.  
  
Task 1.2 I'm assuming is related to the matrix material in the fiber-matrix network scale, at present it is incorporated via direct finite element modeling, but I am unfamiliar with the rule of mixtures so I am uncertain as to our completiong of that portion of the task, though if it is related to Dan's implementation of fiber-matrix-parallel which is a modification of the fiber-only network scale code, then perhaps we have already completed it.  
  
Task 1.3 is the task we have spent most of our efforts directly on at the moment, with Dan's work on migration and load-balancing which will underly work on multiscale adaptation. Developing the adatation algorithms which will actually provide the functionality required for 1.3 has not even begun however, and since at present we have only incorporated the tissue and networks scales into the simulation, and not the cellular scale, the algorithms we develop will be a large step toward this but will not strictly speaking complete 1.3.  
  
  
Specific Aim 2. We will extend our initial model of failure to include progressive damage and damage to the   
nonfibrillar components of the tissue  
  
Task 2.1. We will implement a model of progressive failure in fibers.   
Task 2.2. We will implement a model of interfibrillar material failure.   
Task 2.3. We will perform a series of experiments to specify and test the model, including multiaxial loading.  
  
None of the tasks in specific aim 2 have been approached yet, at least not by either Dan or myself. AMSI should currently be able to support all of these goals, unless one of them prevent a particular multiscale challenge I'm not currently able to envisage.  
  
  
Specific Aim 3. We will incorporate viscoelasticity into the model via fibers, interfibrillar material, and cells.   
  
Task 3.1. We will add viscoelasticity of the collagen fiber to the model.   
Task 3.2. We will add viscoelasticity of the cells to the model.   
Task 3.3. We will add viscoelasticity of the interfibrillar material to the model.   
Task 3.4. We will combine the various viscoelastic effects with a biphasic model for the interfibrillar material,   
allowing us to account for water motion in the tissue.   
Task 3.5. We will perform a series of experiments on engineered constructs to assess the validity of the model.  
  
Similar to specific aim 2, none of these tasks has been approached yet, and AMSI should posses enough functionality to facilitate all of them.

**Specifics regarding implementations of data removal, data addition with load balancing, and migration**

Data removal

A function called RemoveData has been added to the control service class. This function takes a vector of the indices of the data to be removed and the id of the communication pattern. The sender task provides a vector of indices, the receiving task provides an empty vector which will be filled. So, it is assumed that the sending task decides on the data removal. If it is the receiving task that should decide data removal, the user should then be able to simply create a reciprocal comm pattern and call data removal using that instead.

Currently, the data distribution itself is not affected, but rather the communication pattern is directly edited. This should be fixed so that the data distribution (the lower level structure) is also updated. Each sending task process uses the comm pattern and removal indices vector to incrementally reduce elements of the comm pattern matrix and accumulate a list of removal indices to send to the relevant receiving task processes. Using PCU, each receiving task process only receives removal indices from appropriate sending task processes. Each receiving task process updates the affected elements of the comm pattern and offsets the removal indices accordingly. This final list of removal indices on each receiving process is sent back to the user to update data outside of AMSI. There is no guarantee that the removal indices returned by AMSI are ordered.

Each sending task process updates their local portion of the comm pattern during the data removal, and so comm pattern assemble is called with RemoveData for the sending task so that all sending task processes have the entire and correct comm pattern. Each receiving task process updates their local portion of the comm pattern but the receiving task is not assumed to have a full version of the comm pattern on each process.

Data addition and load balancing

A function called AddData has been included in the control service class. There are three inputs to the function: a vector of indices, communication pattern id, and a list or vector of objects. There is some overkill in the inputs which should probably be addressed. The function returns a comm pattern id for initialization of the new added data.

The sending task inputs the number of data to be added, the comm pattern id, and a list of objects corresponding to the data distribution associated with the comm pattern. The list of objects should contain the new added data at the end. The task calls a load balancing function which updates the comm pattern and an initialization comm pattern. The objects list is reordered based on the new comm pattern. Indices are not sent to the receiving pattern, instead the new comm pattern and initialization comm pattern are simply reconciled across the tasks.

The initialization communication pattern defines the difference between the old comm pattern (before the addition) and the new comm pattern (after the addition). This is so that the user can pass initialization data, if need be. This additional comm pattern is actually created when the original comm pattern is created, i.e. the functions createCommPattern and RecvCommPattern have been updated to create two comm patterns. Right now, this is automatic but probably should be optional. A mapping has also been added which finds the initialization comm pattern from the original comm pattern.

The receiving task inputs a list of indices to be filled, the comm pattern id, and a dummy list of objects that is not used. The receiving task first receives the new comm pattern and initialization comm pattern via the reconcile comm pattern function. It then fill the indices list based on these comm patterns.

A function called CommPattern\_UpdateInverted has been added to the control service class (this is not unique to adding data, but for after data has been added and removed). An option to construct an inverted comm pattern is provided in the function CommPattern\_UseInverted, since this functionality is presumed to be common in multimodel simulations, i.e. the transpose of the comm pattern is used to send data in the opposite direction. The new added function simply updates the inverted comm pattern with the new values of the original comm pattern after removing and adding data.

(Note: The user will presumably call both RemoveData and AddData. The user should finish all data manipulations regarding the first call before starting the second. It is suggested, as noted above, to call RemoveData then AddData, but the opposite should work.)

Two default algorithms for the load balancing of new data have been added to AMSI. The AddData function should be updated so that the user could choose between them or provide their own. These default load balancing algorithms assume all data corresponds to equal workloads. One spreads the new data evenly across the comm pattern with no regard for current workloads. The other fills the least loaded receiving task processes first.

Data migration

Migration is split into three functions: planMigration, migration, and shareMigration. The first two only communicate within the receiving task associated with the comm pattern passed to the functions. The last communicates between both tasks on the comm pattern.

The planMigration function is mostly a wrapper for a default planning choice provided in AMSI or for a routine the user provides. This functionality still needs to be implemented. For now, a default planning routine is called that is meant to fully test the other migration routines. This routine is not meant to be a good migration routine. Like the load balancing routines, the default migration routines provided in AMSI will not know better than to treat all data in the comm pattern as having equal associated workloads. Regardless, this routine fills a vector with what data to send which process, for each process.

The migration function actually sends data between processes on the same task according to the information returned from planMigration. The user should collect any needed information and add it to the buffers to be sent during the migration. A process that receives a piece of data will know which process it came from, its old index in its previous element in the comm pattern matrix, and which sending task process it is associated with. After the migration call the user can use the received data to reconstruct the associated work on the new receiving task process.

Finally, the shareMigration function communicates changes to the data ordering to the sending task. Each receiving task process which received migration data communicates information to the associated sending task process for each data received. This information is the old and new receiving task processes and the index within the old element of the comm pattern matrix. This is enough information for the sending task processes to reorder their data respectively and update the comm pattern. The final step in this function is reconciling the comm pattern. After and outside this function the user, as with adding data, can call CommPattern\_UpdateInverted if an inverted comm pattern is being used.