18.337 Final Project
Final Report
Andrew Seagraves

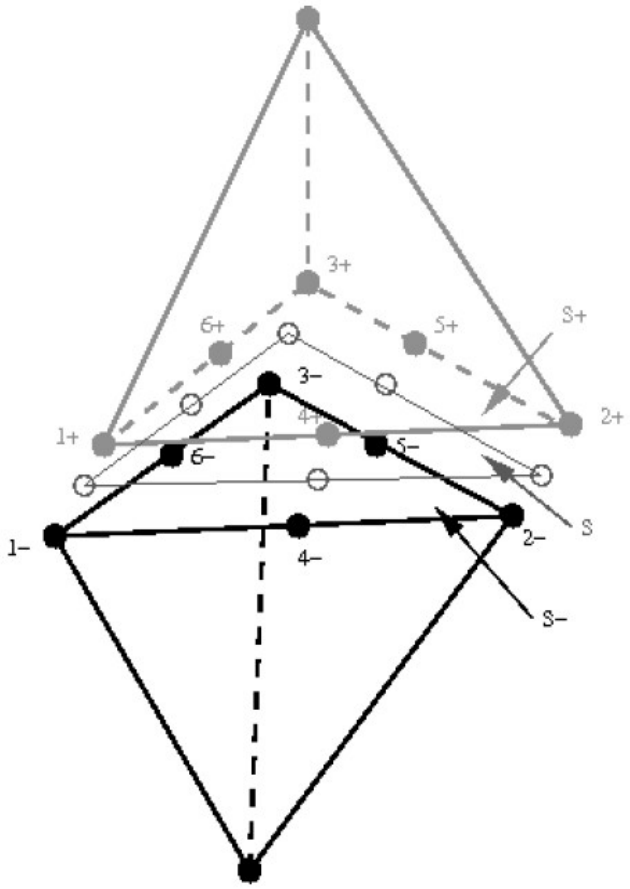
Progress Towards a More Efficient Initialization for Discontinuous Galerkin FE Codes Based on Interface Elements

Abstract

In this final report, I will describe my progress towards developing a new initialization algorithm for my parallel discontinuous Galerkin finite element research code. First, some of the necessary background information concerning the parallel DG initialization is reviewed. Then the basic steps required by the parallel DG initialization code are discussed. Then a new fast serial initialization algorithm, developed previously by my advisor, which avoids many of the problems of the current algorithm, is described. Next, the work I have done in extending this algorithm to the parallel case is discussed. While the code that I have developed successfully completes the first several steps of the parallel initialization, based on the ensuing discussion of this code, it will be concluded that the last 2 steps of the initialization present a highly non-trivial (and previously unforeseen) challenge in algorithm development. A possible brute force approach to complete the last steps of the initialization is described and the relative costs and benefits of this approach are weighed. Finally, some future directions of work towards developing a complete parallel initialization algorithm, are outlined.

Introduction

The goal of my project for this course has been to improve the initialization portion of the finite element code that I use for my graduate research. The current version of the code that I am using is an explicit parallel Lagrangian finite element solver, based on the spatially discontinuous Galerkin framework [1], which is capable of modeling dynamic fracture in nonlinear elastic materials (i.e. brittle materials like glasses or ceramics) [2]. The particular DG formulation that we employ, which was developed previously in our group (i.e. see [1]), computes essentially the same solution as a continuous Galerkin (CG) finite element method, except that the continuity at interelement boundaries is enforced in a weak sense (whereas in the CG framework, this continuity is enforced strongly and adjacent element facets remain coincident throughout the simulation). Hence, for the DG framework, the displacement fields are allowed to be discontinuous across the interelement boundaries. For the formulation to be consistent with the variational statement of the continuum mechanics problem, additional interelement integrals are required (in addition to the typical integral equations which are solved in the CG framework) which account for the proper transfer of momentum between the discontinuous elements. The contribution of these integrals to the total discretized system of equations is computed and assembled through the use of interface finite elements, which consist of two quadratic triangular surface elements whose nodes coincide with those of two adjacent element facets (depicted below).



Since displacement discontinuities are allowed at all interelement boundaries in the DG framework, the entire mesh must be discontinuous (i..e. None of the elements are connected to any other elements so that every element possesses its own unique nodes) and interface elements must be created prior to the calculation at every interior element boundary in the body. For a serial calculation, this insertion can be done in a fast and straightforward manner using the following two steps:

The Required Steps in a Serial Initialization Algorithm

1. Starting with a CG mesh, first break up all the volumetric elements within the processor, so each element possesses its own unique nodes. This requires the creation of a new connectivity array for the elements, which contains the new node numbers for each element once the mesh has been broken up. This can either be done in an incremental fashion by sequentially modifying the existing connectivity table for the original CG mesh, or by creating the new connectivity table from scratch. The former approach, which was the approach adopted by my Post-Doc who wrote the current initialization code, is extremely slow. This is due to the expensive data structures used and the incremental nature of the algorithm. The latter approach, of creating the new connectivity from scratch (without the need for any additional costly data structures), is the approach taken by my advisor in the improved sequential initialization algorithm that he provided me with. In either case, once the mesh has been completely broken up by adding all the new nodes, the various solution arrays (displacements, velocities, etc.)

stored at the nodes must also be reallocated. In the case of my advisor's new algorithm, this reallocation is done all at once, after all the new nodes are created. On the other hand, my Post-Doc's algorithm does this reallocation incrementally when each new node is added to the mesh.

2. Next, interface elements are "inserted" at all the interior facets within the processor, with the facets on the boundary of the processor ignored (since in a serial calculation, the boundary of the processor is the global boundary of the body). The insertion of an interface element requires the allocation of additional space in the solution arrays to hold data for each quadrature point of the interface elements. It also requires that a connectivity table for the interface elements be established and be merged with the original connectivity table established in step **1** for the volumetric elements. The current algorithm developed by my Post-Doc completes this step by incrementally adding the connectivities of the interface elements to the existing connectivity table for the volumetric elements. On the other hand, my advisors new serial algorithm creates a separate connectivity table for just the interface elements, and then once that array is completed, it reallocates the total space needed for the combined connectivity tables, and then it merges them.

In some test runs for the two approaches just described for initializing a mesh on a single processor, I have found that my advisor's new serial algorithm is ***several orders of magnitude faster***. Based on the fact that it provides a large speed-up in the serial case, I originally proposed trying to extend my advisor's algorithm to the parallel case as my final project for this course.

Some Background Aspects of the Parallel Initialization for DG

In the parallel initialization case, it turns out that things are much less straight-forward than in the serial case. This is due to the fact that interface elements must be not only be created at all the interior facets within each processor (something that can be done in a completely data-parallel fashion with my advisor's new algorithm), but there must also exist some facility for creating the interface elements at the processor boundaries which are still interior to the global boundary of the solid body. In the context of our DG code, I found that the solver requires that the full interface element must be created in only one of the two processors which share an internal boundary of the body. **Arbitrarily, the original developer of the code decided that the full interface element must exist in the processor with the lower processor Id.** This is depicted schematically in the figure below.

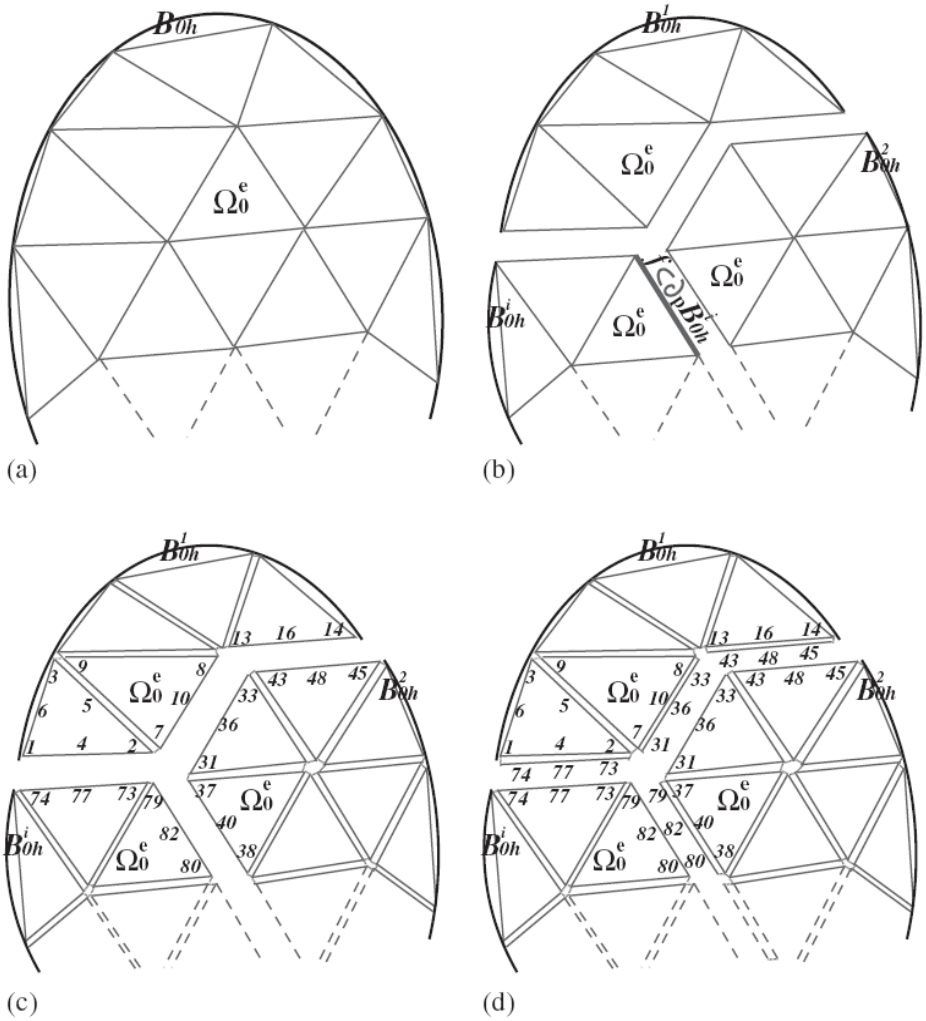


Figure 3. Creation of the partitioned discontinuous mesh (schematic): (a) initial discretization; (b) partitioned mesh; (c) interfaces inside partitions; and (d) interfaces between partitions.

The key thing to note from the figure above is the global numbering of the nodes in part (d) of the figure. Examining the part (d) above, we see for example, that processor #1 is neighbors with processor #2, and so processor #1 must create the full interface element at all the facets geometrically matching those in processor #2. Now, examining the first figure introduced of a single interface element above, we find that since processor #1 contains the full interface elements which must exist between the two processors, it must also contain the nodes from processor #2 corresponding to the opposing halves of the interface elements. Hence these boundary nodes in processor #2 must also live in processor #1. This is indicated in part (d) of the figure above through a global numbering of the

nodes as shown.

Communication Maps: A C++ Object For Dealing with Shared Nodes

The fact that these processor boundary nodes must coexist in two different processors is not a unique aspect of a parallel DG calculation. In fact, for a parallel CG calculation, all nodes matching geometrically at the boundaries between two processors must coexist in both processors since the mesh is not broken up. Hence in the case of CG, a node on the boundary of a processor can also exist in many different processors. In order to compute the correct finite element solution in parallel CG, each processor must send information to and receive information from its neighboring processors for each of the shared boundary nodes so that every processor can assemble and solve a set of equations independently which will return the consistent (and correct) finite element solution at the shared nodes (hence the displacement of the shared nodes computed independently in each processor will be the same). In order to send the necessary information to another processor on the shared boundary nodes in our finite element research code, a C++ object called the **"Communication Maps"** is defined in each processor. One communication map object is defined for each shared processor boundary and returns the following information:

- The rank Id of the neighbor processor
- The total number of nodes in this processor which are shared by the neighbor
- A list of the local node Ids defined in this processor for those nodes which are shared by the neighbor.

In each corresponding neighbor processor, the same information is stored in a Communication Map object, with the only difference being that the list of the shared nodes reflects the corresponding local Id of those nodes in that processor. The key point is that the ordering of the two lists with respect to the shared nodes is the same, even though the local Ids for those nodes have been assigned independently in the two different processors. The Communication Map object enables the efficient sending and receiving of information on the shared nodes at each time step, as mentioned previously, because the ordering of the list of local node Ids is the same in the two processors. Hence at each time step, the

Communication Map object allows you to construct send buffers and do MPI message passing in a completely automated fashion, as the neighboring processor Id is known, the local Ids of the relevant nodes involved in the message are known, and the ordering in which the information should be assembled in the send buffer is known. Similarly, each processor uses the Communication Maps to receive messages on their shared boundary nodes in an automated fashion given that they know which neighboring processor Ids to receive from, they know the sizes of the buffers they will be receiving, and they know to which local node each piece of information in the receive buffer should be assigned (given that the ordering of the local Ids is the same in the two processors).

Similar to a parallel CG calculation, for parallel DG, there will also be nodes shared between adjacent neighboring processors. However, as mentioned previously, these shared nodes arise from inserting the interface elements at the processor boundaries in the lower Id processors. Hence, in the case of a DG initialization, the communication maps which were created for the original partitioned CG mesh must be recreated to reflect the new nodes which are connecting adjacent processors once the mesh has been completely broken up and the interface elements have been inserted at the processor boundaries.

Furthermore, every node across every processor is also required by our solver to have a unique global Id. This implies that for each new node created in the processor boundary interface elements, the correct global Id of that node must be transferred from the adjacent processor in which that node was originally living. This is depicted in the parts (c) and (d) of the figure above.

Winged Facet: A C++ Object Serving as a Facet Data Structure

Now since I have introduced and discussed the C++ object called the Communication Maps, I will describe another important C++ object which is created and used in our ***parallel CG*** research code to describe the facets in a mesh called the **Winged Facet**. This C++ object contains the following

information: For each pair of two adjacent tetrahedra interior to a processor, the Winged Facet is a container which provides a pointer to the element numbers of the tetrahedra that are on either side of the element interface (with a convention that labels the adjacent tetrahedra as either top or bottom). If the facet lives on a processor boundary, it is only adjacent to one tetrahedra. In this case, the Winged Facet points to only that one tetrahedra, while the pointer to the other side is NULL. This allows the code user to distinguish between facets which are internal to the processor and facets which are on the processor boundary. The Winged Facet objects also return the six local node numbers associated with each facet (note that in the CG case there are always six nodes corresponding to each facet whether it is internal, or on the boundary). Note that I am introducing and explaining the concept of the Winged Facets (a data structure used for a CG calculation) at this point because my advisor uses them as a tool in his algorithm for initializing a DG mesh on a single processor, and so they will be relevant later.

The Required Steps in a Parallel DG Initialization Algorithm

First we have information that is known A Priori from the parallel CG mesh partitioning which is always done first:

- First, a CG mesh file containing the coordinates of each node and the nodal connectivity of each element is loaded into each processor and the mesh information is partitioned across the processors by having each processor keep information only on its relevant set of geometrically-connected elements. The arrays necessary for storing all of the problem unknowns (displacements, strains, stresses, etc.) are created within each processor for all of the nodes within the processor. Furthermore, the old Communication Maps, and the old Winged Facets have been created and are available in each processor. At this point, the code is initialized sufficiently for doing a standard parallel CG calculation.

Then we must take the following steps for parallel DG initialization

(1) Starting with the partitioned CG mesh, first break up all the elements within each processor, so each element possesses its own unique nodes. This requires that the nodal connectivity be modified for each element within each processor to reflect the addition of new nodes to the mesh. Furthermore, it requires that additional space be allocated in the various solution arrays for each additional node created. Note that at this point, each element belongs uniquely to only one processor since the whole mesh has been broken up.
(2) Next a unique global ID is assigned to each node in the mesh across all the partitions. The function to complete this step already exists and I will not modify it.
(3) Next, interface elements are “inserted” at all the interior facets within each processor, with the

facets on the boundary of the processor initially ignored (at this point each processor cannot yet distinguish between whether facets on its boundary are shared by another processor and thus are interior, or are located on the external boundary of the body). The insertion of an interface element requires the allocation of additional space in the solution arrays to hold data for each quadrature point of the interface elements. It also requires the creation of a connectivity array for the interface elements.

(4) Next, some operations are required within each processor to determine which element facets on its boundary have neighbors in other processors with higher processor Id and thus also require the insertion of interface elements. The processors with lower rank Id must create new nodes on their boundaries corresponding to these interface elements. The connectivity of these boundary interface elements must be added to the total connectivity.
(5) The correct global Ids for the new nodes created in step **4** must be transferred from the processors with higher rank Id to the processors with lower rank Id where these nodes have been added at the processor boundary.
(6) The old Communication Maps must be recreated to reflect the newly shared nodes resulting from the interface elements being created between processors.

A detailed and complete explanation of how the current (very slow) parallel DG initialization completes steps (**1**)-(**6**) can be found in my earlier Progress Report for this project, so I will not go into detail about that here. Instead, now I will describe my progress in attempting to extend my advisor's fast serial algorithm to the parallel case.

Progress Towards Developing a New Parallel DG Initialization

My advisor's serial algorithm, as mentioned previously, is able to complete the serial steps (**1**) and (**3**) above in a very small amount of time (less than one minute) for meshes that are approximately 50,000 elements. Based on the success of this algorithm at initializing on a single processor, my advisor told me that he wanted me to try and extend his algorithm to the parallel case. In principle, he argued that there should be enough information available in each processor to complete step (**4**) without doing any communication. Then, he argued that each processor should have enough information to complete either one of steps (**5**) or (**6**) in serial (i.e. without communication) and then complete the last remaining step with some communication between the processors.

Indeed he was correct that his algorithm could be extended to complete step (**4**) without communication as I have successfully modified his algorithm to do so. However, by choosing to

design the code so that step (**4**) is done in a serial fashion, I have found that the second of his assertions is totally incorrect. Indeed, as I will reveal in the following discussion, under this coding paradigm, both steps (**5**) and (**6**) not only require lots of communication, but also could require a brute force approach using full range queries on top of that (i.e. matching nodes between processors using the coordinates, which is extremely expensive). Now I will discuss all of this in detail:

Step (1)

My advisor's algorithm starts with only the old Winged Facets, the old nodal connectivities, and the old coordinates arrays inside of each processor. The algorithm breaks up the mesh by assigning to each element new unique nodes and adding the new node numbers to a new, pre-allocated connectivity array for the volumetric elements. On top of this, an array is populated which serves as a mapping from the new local node numbers to the old local node number. In other words, when this algorithm has finished breaking up the whole mesh in step (**1**), I have access to an array inside of each processor called newToOldId[], which takes as an argument the new node Id, and gives the old node Id (i.e. Old_node_Id = newToOldId[New_node_Id]).

Step (2)

Since the mesh has been completely broken up, in the parallel case I would complete step (**2**) by simply calling the pre-existing function which renumbers the global Ids of each node for a discontinuous mesh.

Step (3) Serial Case

My advisor's algorithm completes step (**3**) by cycling through all the Winged Facets defined in the mesh and checking if the Winged Facet points to two adjacent tetrahedra. If so, the facet is internal to the mesh and thus requires insertion of an interface element in the serial case → for each of these

qualifying facets, a counter is incremented. When the loop finishes, the total number of interface elements that must be created on the interior of the processor has been counted. In the serial case, this is the total number of interface elements that must be created since there are no interface elements living at the boundary. The total number of interface elements is then used to pre-allocate an array which holds the interface element connectivity. Then the algorithm again cycles through all the Winged Facets, checking for the same condition as before. If it is satisfied, the facet is internal, and so the algorithm creates the interface element between the two tetrahedra. Note that since these are the old Winged Facet objects, they still contain the local Ids of the six old nodes which used to populate that interior facet in the parallel CG case. However, from the new to old mapping, and the new connectivity array established in step (**1**), the twelve new nodes which now populate that facet can be quickly retrieved and added to a pre-allocated interface element connectivity array.

Steps (3) and (4) Combined for the Parallel Case

My modification to this algorithm (representing my main coding contribution of this project), allows the algorithm to complete step (**4**) essentially in tandem with step (**3**). The algorithm first cycles through all the Winged Facets and does two things: 1-It increments a counter for every Winged_Facet, thus counting the total number of facets in the mesh, and 2-It checks whether each facet is internal to the body, and if so, increments a second counter each time a facet is found (the value of this counter after the loop terminates represents the number of internal facets which require interface elements, and obviously does not include the number of boundary facets which require interface elements). Given that the total number of facets is now known, I create an array of that total size, which will eventually hold a flag for each of the Winged Facets. This flag array will be populated in the following to indicate simply whether or not that facet lives in a neighboring processor with a higher processor Id, and thus is a boundary facet which requires the insertion of an interface element.

The modified algorithm then cycles through the Winged Facets again, but this time, checks to see if the facet is on the external boundary of the processor. If this is the case, then the facet may either be on the external boundary of the body, or it may reside in a processor with a higher or lower processor Id. At this point we need to determine if this facet lives in another processor with a higher processor Id. Since we are cycling through the old Winged Facets, I use them to recover the six old node Ids for that boundary facet. Next, I search through each of the old Communication Maps for neighboring processors with higher Id which, as mentioned previously, are simply lists of the old local node Ids populating those boundaries. If I am able to locate all six old node numbers for the Winged Facet inside of one of the Communication Maps for neighbors with a higher processor Id, then this Winged Facet requires the insertion of an interface element. In this case, the flag array for this facet is assigned the Id of the higher neighboring processor. Also, the counter which originally stores the number of interior interface elements is incremented to reflect the additional interface element which will be added at this processor boundary.

Otherwise, if the facet is external and all of the Communication Maps for higher rank neighbors are searched without identifying all six nodes of the Winged Facet, then this facet exists either on the global external boundary of the body, or in a processor with a lower rank Id. In this case, or the case that the facet is internal, the flag array for that facet is set to -1, indicating that this is not a facet which requires an interface element at the processor boundary.

When this part of the algorithm has finished the flag array has been completely populated and the total number of interface elements to be added is known. The latter of the two pieces of information is then used to pre-allocate the interface element connectivity array to reflect the full number of both interior and boundary interface elements in the parallel case. The next part of the algorithm, which I left

completely unmodified, inserts the interface elements at all the interior facets inside the processor. The last step is to insert the interface elements at the processor boundaries. This is done by cycling through all the Winged Facets and checking if the value of the flag array for that facet is > -1. If so, the algorithm grabs the six existing local node numbers already living on that facet, adds them to the interface element connectivity, and then fills in the other side of the interface element connectivity with distinct new local node numbers. From this process, the total number of new nodes to be added to the solution arrays and to the coordinates array can be easily counted.

After this loop exits, the total interface element connectivity in each processor has been assigned, and the number of new nodes to be added to the mesh has been determined. Given the latter piece of information, the various solution arrays and the coordinates array are reallocated to reflect addition of these new nodes.

At this point steps (**3**) and (**4**) are complete!

Steps (5) and (6) (Where I Am Currently Stuck)

Given that I have chosen to design the modified algorithm to complete steps (**3**) and (**4**) in a serial fashion (as suggested by my advisor), a paradox arises which makes the completion of steps (**5**) and (**6**) extremely difficult and most likely very expensive. At this point in the initialization, the global Ids of the newly created nodes on the processor boundaries are not known and must be transferred. In principle, the only facility for transferring them from the higher processor to the lower one, is to send them between the processors using the Communication Maps describing that processor boundary. However, the new Communication Maps are not available in any of the processors at this point to be used for this purpose. So there is no facility available for transferring the global Ids.

Conversely, imagine trying to complete step (**6**) after completing steps (**3**) and (**4**). In principle, the

only way to recreate the Communication Maps in a straightforward manner would be to determine the list of new local node numbers in each processor which also reside in a processor with a lower Id. This list of nodes would be added in the new Communication Map with some arbitrary ordering. Then the global Ids corresponding to that particular ordering would be sent to the lower Id processor. The lower Id processor would receive this list of global Ids and would produce the correct Communication Map with its own local node Ids in the correct ordering, corresponding to the global Ids just received. However, the global Ids of these specific new processor boundary nodes are not known yet after completing steps (**3**) and (**4**) in the lower processor (they are supposed to be transferred somehow in step (**5**)). Hence, the only straightforward way of recreating the Communication Maps requires that the global Ids have already been transferred. But, as I just discussed, the global Ids cannot be transferred in any straightforward manner without the newly recreated Communication Maps.

Hence, a paradox arises which makes the completion of steps (**5**) and (**6**) actually quite difficult to do in practice. Since I arrived at this realization in the last few days, I have been able to think of only one possible algorithm that would be able to complete steps (**5**) and (**6**), given steps (**1**)-(**4**). The algorithm would work in the following way:

- In each processor scan through the Winged Facets and locate all of the facets which also belong to processors with lower Id
- Construct lists of all the Winged Facets living in each lower neighboring processor
- For each Winged Facet list corresponding to each lower processor, construct an array which contains the coordinates for each of the nodes on each of the Winged Facets. This array would be of size spatial_dimension * number_of_Winged_Faces * 6, and each 18 consecutive entries would correspond to the x, y, and z coordinates of all the nodes on each Winged Facet.
- For each array just constructed, construct an additional array where each of the x, y, and z triplets is replaced by the global Id
- Also in each processor, prepare a receive buffers of the appropriate size which will contain the received coordinates and the received global Ids from the higher neighboring processors
- Send the coordinate arrays to the lower neighboring processors
- Receive the coordinate arrays in the corresponding lower processors
- Search through the coordinates of each of the boundary Winged Facets to find their geometric match in the receive buffer.
- When each Winged Facet is located in the receive buffer, add the newly created local Ids of the interface elements into an array which matches the ordering of the coordinates array just

received for the Winged Facet that has just been matched geometrically.

- Send the global Id arrays to the lower neighboring processors
- Receive the global Id arrays in the in the lower processors
  The ordering of the global Ids matches the ordering of the coordinate array, so it also matches the ordering of each new interface element Id array that has been created in this processor.. Hence, the new global Ids are now known in all the processors for the newly created boundary nodes. Given this fact, the Communication Maps can be reconstructed as described previously from the newly transferred global Id.

Hence, the only algorithm that I have been able to envision is quite complicated, requires a lot of communication between the processors and, furthermore, requires that coordinate searches be done in all of the processors. As is well known, coordinate searches can be extremely expensive and could possibly be a major bottleneck in this new parallel DG initialization algorithm.

Summary and Future Work

For this project, I was concerned with developing an faster initialization module for my DG finite element research code. Based on the good performance of an alternative serial initialization algorithm developed my advisor, we decided that I would try and extend this serial algorithm directly to the parallel case. While I was able to extend his algorithm to complete step (**4)** in the DG initialization, I've found that unforeseen complications arise in steps (**5**) and (**6**), due to the paradox that step (**5**) must be done to complete step (**6)** in a straightforward manner, while step (**6**) must be done in order to complete step (**5**) in a straightforward manner. Indeed, the only algorithm that I have been able to conceptualize which will complete steps (**5)** and (**6**) given the code I have already developed to do steps (**1)** through (**4**) requires a lot of additional message passing. Even worse, it requires coordinate searching which is a notoriously expensive process. Based on the result of this project, my advisor and I are going back to the drawing board and we are trying to work out whether it is indeed possible to complete steps (**5**) and (**6**) in the initialization without doing coordinate searches. If it is possible, I will likely continue to develop my current code, completing steps (**5**) and (**6**) without coordinate searches. Otherwise, if coordinate searches prove to be inevitable under the current framework that I have developed, we will either continue in that direction and I will implement the code outlined above, or we

will start over completely. I am really hoping that the former is the case given all the work that I have done up to this point! In conclusion, I should note that my even though I have not completed a full implementation of the code, my advisor is satisfied with my progress in understanding the current code and in partially developing a new initialization code.

References

1. L. Noels and R. Radovitzky, "An Explicit Discontinuous Galerkin Method for Non-Linear Solid Dynamics: Formulation and Scalability Properties," *International Journal for Numerical Methods in Engineering*, **74:** 1393-1420.
2. A. Seagraves and R. Radovitzky, "Advances in Cohesive Zone Modeling of Dynamic Fracture," Dynamic Fracture of Materials and Structures, Springer Science and Business Media, In Press.