Dear CTTS software and algorithms team:

The question of making a thread safe version of the mesh level operations came up in one of the recent calls with the initial area of concern relating to ensuring that the assembly of the element stiffness matrices and load vectors into the global system would be thread safe. The RPI team has been looking into this issue and has identified three approaches, each of which can have its pros and cons. In addition, we have also identified some issues with respect to interacting with the solvers as well as a couple of options with respect to implementation. The remainder of this email will summarize what we have determined and request the involvement of the appropriate team members for moving forward on doing this job well. (Note much of the discussion here is pretty high level and not overly precise, but hopefully is a useful starting point.) It should be noted that we do not have lots of experience in this specific area, so those that know more can feel free to point out anything they see that is known to be a bad idea.

I suggest that we use email to share some ideas and options. If this process yields a clear approach along with a distribution of who will be doing what with whom, we will just proceed. If after some back-and-forth via email, we are not clear on next steps, we will set-up one or more conference calls.

Approaches to doing thread safe assembly

Independent sets or coloring: In this process there is a processing step that is carried out as needed to determine sets of elements that do not share dof with each other, thus there would be no conflicts when assembling terms for the parallel process of such sets. Although this approach requires steps of processing to determine the sets, it could be the most efficient approach. Also of importance for other operations where the data layouts are altered as part of the operation (such as mesh adaptation), it is the only reasonable approach (as far as we can see).

Using Atomics to block when needed: This could be the simplest to implement, but could be inefficient if there are lots of conflicts in adding data into the matrix. For matrix assembly, it may not be hard to do something simple that keeps the number of conflicts low.

Independent sub-assembly followed by full assembly: In this case extra memory is employed to collect values that is processed on a thread level and then they are assembled. If I understood correctly, this can be done using on the order of 2 times the memory of a single copy of the system. For the case of the global system matrix, this would appear to be a bad idea. If the understanding is wrong and it requires very little extra memory, or the extra memory is deemed to be acceptable, this is not hard to implement.

Issues with Solvers

Based on feedback we have received from PETSc experts, even if we process independent sets of elements via parallel threads (which is thread safe), it is not clear that PETSc processing will be thread safe. This may be avoidable if we directly define the PETSc matrix and load vector external to PETSc and then pass them to PETSc in serial on process (without threads). Clearly this is a sub-optimal solution. Given those concerns, we have also started to look at use of STRUMPACK and Hyper. Although Hyper has more emphasis on threading than PETSc currently does, it was not fully apparent on a quick review if it would be easy to do what was needed.

Considering CTTS plans to work with Sherry on using STRUMPACK, we would like to work with her to examine the options for doing an efficient thread safe assembly. We can start with a process of building the CSR matrix external to STRUMPACK and passing it in bulk, which our review of the documentation indicates should be possible. This will have the loss of thread level parallelism. We also noticed in the STRUMPACK documentation that the matrix level reordering procedures were a bit of an issue. We would be interested in investigating the potential of us basically determining dof ordering directly base on mesh information so the CSR matrix structure we send is already good. We would start from work we have done on improving mesh partitions that balance elements based on either graph partitioners, or recursive bisection, to balance the dof per part accounting for keeping the partitions well shaped. (This is done by carefully considering the mesh entities that hold dof's, etc. – we can send you the paper on it.)

Question of implementation

Unlike internode communication support where the implementation will be based on MPI. We have multiple options for the dealing with threading. Considering what is already done in M3D-C1 and STRUMPACK, OpenMP is a natural choice. However, if we want to also deal in the future with GPU's this may not be ideal. We have been doing work with Kokkos which is suppose to make it easy to do many core and GPU-based systems. It is possible to uses both Kokkos and OpenMP in the same code as long as they do not interfere. The OpenMP is M3D-C1 looks to just be in some tight element calculation loops and it would seam that as long as we do the bulk passing of the matrix to STRUMPACK, there would also not be a problem there. We would very much like to hear from others that may know more about these things to share their knowledge and opinions about the implementation options, their relative complexity and efficiency of their operations.