Parallel programming for Finite Element Analysis - a step-by-step guide

Chennakesava Kadapa, c.kadapa@swansea.ac.uk

Swansea Academy of Advanced Computing, Swansea University, Fabian Way, Swansea SA1 8EN, Wales, UK.

Abstract

We present a detailed step-by-step description of parallelisation for the finite element analysis for distributed-memory based high-performance computing architectures. The parallelisation is based on the standard message passing interface OpenMPI and the opensource library PETSc for parallel matrix solvers. Code snippets are provided for programming in FORTRAN and for OpenMPI and PETSc subroutines. Computation of data structures is descirbed using simple test cases for Poisson equation and linear elasticity in two-dimensions. Scalability studies are presented for Poisson equation, linear elasticity and incompressible Navier-Stokes.

Keywords: Finite Element Analysis; Petsc; Iterative solvers; Parallelisation; OpenMPI; FORTRAN

1. Introduction

Developing a parallel code for the numerical schemes is always a challenging task. There are numerious difficultires associated with parallelisation and these difficulties depend upon the type of problem to be solved, the size of the problem to be solved, the underlying numerical scheme and the computing architecture to be used.

2. PETSc

PETSc is an open-source library that is built primarily on top of the standard message passing interface, OpenMPI, and offers various data structures and subroutines, as depicted in Fig. 1, for simulating large-scale numerical models on high-performance computing architectures. Though PETSc was initially developed in C languague, today, it offers support for C++ as well as FORTRAN languages. The sub-libraries in PETSc provide different levels of abstraction suited to the requirements of the user. With numerous built-in parallel iterative solvers, preconditioners and wrappers for various third-party libraries, PETSc provides a powerful platform for developing parallel finite element codes for high-performance computing systems.

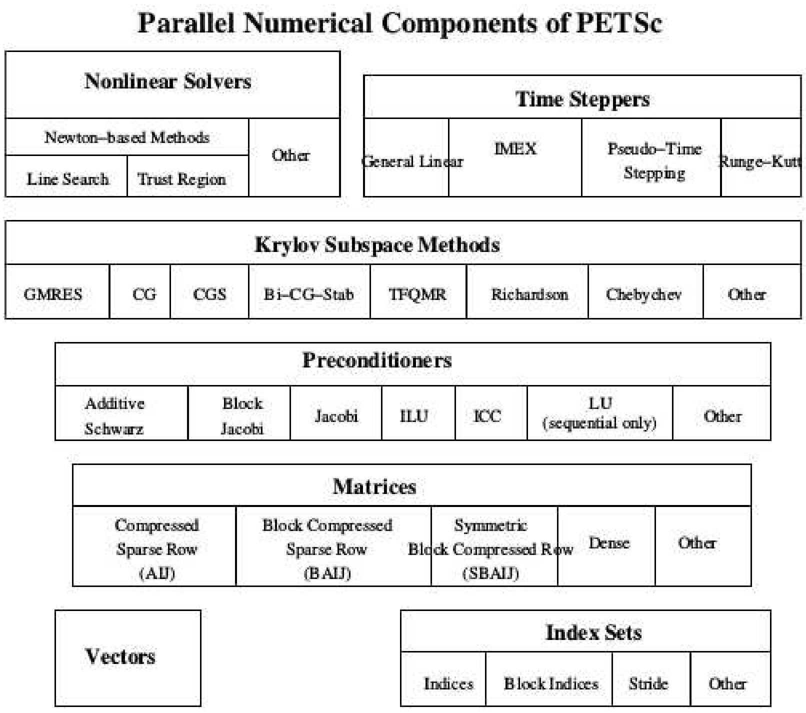

Figure 1: Numerical libraries of PETSc

3. Parallelisation: step-by-step procedure

The procedure for parallelisation of a finite element code depends upon the computing architecture to be used, i.e. shared-memory or distributed-memory or graphics processing unit (GPU) architecture; the procedure varies considerably depending upon the computing archtecture of interest. While the implementation for shared-memory architecture is relatively easy when compared with that of distributed-memory architecture, literature and our experience suggests that shared-memory parallelisation scales poorly when compared with the distributed-memory parallelisation. In this article, we present the procedure for parallelisation of finite element code for distributed-memory architectures.

In order to understand the procedure and the underlying details of parallel programming for finite element method (FEM), it is important to have a thorough understanding of the procedure for serial FEM. Having a modular design for the serial code helps significantly in parallelising it. The procedure for a serial FEM code (for a linear finite element problem and involvig a single time step (or load step)) can be grouped into five important steps as outlined in Fig. ??. Assuming that sparse matrix is used for the global stiffness matrix, the efficiency of the serial code depends mainly on the speed of matrix assembly (Step 3) and the efficiency of the matrix solver (Step 4). The performance of matrix assembly for sparse matrices can be improved be significantly by preallocating enough memory and by precomputing the matrix pattern before assembling element matrices in Step 3.

Analogous to the serial code in Fig. ??, the important steps involved a parallel FEM code are outlined in Fig. ??. The important difference between a serial code and a parallel code lies in Step 2 and it is this step that determines the overall performance of a parallel FEM code. The way the mesh and the global stiffness matrix are decomposed significantly affects the achievable speedup.

Generation of the sparse matrix pattern for the parallel FEM code requires several data arrays for mapping the node and DOF numbers from a local mesh (or matrix) to the global mesh (or matrix).

Procedure for serial FEM

```
Step 1: Read the mesh
Step 2: Initialise the sparse matrix
        Step 2a: Generate data arrays
        Step 2b: Create the matrix pattern
Step 3: Calculate the global stiffness matrix and the load vector
Step 4: Solve the matrix system
Step 5: Post-process the results
```

Algorithm 1: Procedure for serial FEM.

The steps involved in the parallel finite element method are outlined in Fig. ??. Even though, there are only two additional steps in parallel FEM code, these two steps are the most important and difficult aspects of implementing a finite element code in parallel. The foremost difficulty in parallelisation for distributed-memory architectures arises from Step 2 and Step 3, i.e., the distribution of the mesh among all the processors involved in the communication (Step 2) and reording the nodes and DOFs. This is because the efficiency of parallelisation, often measured in terms of *speedup*, depends primarily on the efficient distribution of elements/nodes in a mesh among the processors. Therefore, creating a partition/decomposition with the best load-balancing over all the processors is extremely crucial. For simple cartesian meshes, the partitions can be created manually. However, for partitioning complex meshes sophisticated partitioning algorithms, such as Metis, Chaco, Scotch or Jostle require to be used. The parallel matrix algebra library PETSc also provides partitioning functionality which internally calls METIS/ParMETIS.

METIS outputs the information for both the elements and nodes in a particular domain, ParMETIS does not. Therefore, using METIS one can avoid complex subroutines for finding nodes that belong to a particular processor.

The procedure outlined here is for when the entire mesh is stored on the every processor in the communication.

Procedure for parallel FEM

```
Step 1: Read the mesh
Step 2: Initialise the sparse matrix
        Step 2a: Partition the mesh
        Step 2b: Renumber nodes and DOFs
        Step 2c: Generate data arrays
        Step 2d: Create the matrix pattern
Step 3: Calculate the global stiffness matrix and the load vector
Step 4: Solve the matrix system
Step 5: Post-process the results
```

Algorithm 2: Procedure for parallel FEM.

variable name	variable type	size	definition
nelem_global	integer		number of elements in the whole mesh
nnode_global	integer		number of nodes in the whole mesh
nodes_per_elem	integer		number of nodes per element
dofs_per_node	integer		number of dofs per node
elem_node_conn	integer array	nelem_global x nodes_per_elem	element-node connectivity
dof_if_prescrided	integer array	nnode_global x dofs_per_node	array for storing whether a dof has prescribed value
dirichlet_data	double array	nDBCsx3	Dirichlet boundary conditions: node, dof, val
force_data	double array	nFBCsx3	Applied nodal forces: node, dof, val
elem_proc_id	integer array	nelem_global	processor id of each element
node_proc_id	integer array	nnode_global	processor id of each node
nelem_domain	integer		number of elements owned by the local/current processor
nnode_owned	integer		number of nodes owned by the local/current processor
node_get_old	integer array	nnode_global	mapping from new node numbers to old/original node numbers
node_get_new	integer array	nnode_global	mapping from old/original node numbers to new node numbers
node_dof_index_old	integer array	nnode_global x dofs_per_node	array for storing DOF values, including any Dirichlet BCs
nodelist_owned	integer array	nnode_domain	list of nodes owned by the local processor
nnode_owned_vector	integer array	n_mpi_procs	array of **nnode_domain** from all the processors
node_start	integer		number of the first node owned by the local processor
node_end	integer		number of the last node owned by the local processor
row_start	integer		numbers of the first row owned by the local processor
row_end	integer		numbers of the last row owned by the local processor
elem_assy_array	integer array	nelem_global x (nodes_per_elem x dofs_per_node)	array for element and global dofs connectivity
solver_to_mesh	integer array	size_global	array for mapping the solution from the solver to the global solution vector

Step1: Read the mesh

Once the mesh is successfully read, we know the following information.

- **nnode_global** — number of nodes in the whole mesh
- **nelem_global** — number of elements in the whole mesh
- **nodes_per_elem** — number of nodes per element
- **dofs_per_node** — number of degrees of freedom per node
- **nDBC** — number of specified Dirichlet boundary conditions

- **dirichlet_data** — data for Dirichlet boundary conditions: <node> <dof> <value>
- **nFBC** — number of specified nodal forces
- **force_data** — data for specified nodal forces: <node> <dof> <value>

Step2: Initialise the sparse matrix
Step2a: Partition the mesh
The fundamental idea behind parallelisation for finite element analysis is to distribute the finite element mesh among all the processors in the communication so that creation, assembly and the subsequent solution of the sparse matrix can be performed in an efficient matter. To achieve this, the finite element mesh needs to be partitioned.

Partioning of the mesh can be done either manually, for simple meshes, or by using tailor-made partitioning libraries such Metis [aa], Chacos [bb] and Scotch [cc] etc. In this work, we use Metis. Upon successful completion, mesh partitioning algorithms in Metis return the following two arrays.

- **elem_proc_id** — processor id for each element in the mesh
- **node_proc_id** — processor id for each node in the mesh

Remark 1: Eventhough, a parallel version of Metis, ParMetis [aa], is available, it comes with a disadvantage that it returns only the **elem_proc_id** array. The user has to compute the **node_proc_id** array. So, it is preferable choice to use Metis instead of ParMetis.
Remark 2: To avoid any unwarranted trouble that might occur if all the processors return different mesh partitions, it is recommended to partition the mesh only on one of the processors in the communication and then brodcast **elem_proc_id** and **node_proc_id** arrays to other processors.

Step2b: Renumber nodes and DOFs
PETSc stores sparse matrices in chunks of continuous rows across the processors. Therefore, to minimise the communication across the processors, nodes (and the corresponding DOFs) in the mesh needs to renumbered in such a way that the nodes local to processor 0 are numbered first, followed by those local to processor 1 and so on.

For renumbering nodes,

i.) Count the number of nodes that are owned by the processor.

```
nnode_owned = 0
Do ii=1, nnode_global
  IF(node_proc_id(ii) == this_mpi_proc ) THEN
    nnode_owned = nnode_owned + 1
  END IF
END DO
```

ii.) Gather the nodes owned by the processor.

```
ALLOCATE( nodelist_owned(nnode_owned) )
ind=1
Do ii=1, nnode_global
  IF(node_proc_id(ii) == this_mpi_proc ) THEN
    nodelist_owned(ind) = ii
    ind = ind + 1
  END IF
END DO
```

iii.) Gather **nodelist_owned** from all the processors in the communication and create **node_num_get_old** and **node_num_get_new** mapping arrays.

```
! Compute 'nnode_owned_vector' by gathering the values of
! 'nnode_owned' from all the processors in the communication
ALLOCATE( nnode_owned_vector(n_mpi_procs) )
CALL MPI_Allgather(nnode_local , 1, MPI_INT,
      nnode_owned_vector, 1, MPI_INT,
      PETSC_COMM_WORLD, errpetsc)

! Calculate the starting and ending node numbers
node_start = 1
node_end   = nNode_local_vector(1)
Do ii=1, this_mpi_proc
  node_start = node_start + nnode_owned_vector(ii)
  node_end   = node_end   + nnode_owned_vector(ii+1)
END DO

! Compute 'displs' array for gathering the node list
ALLOCATE(displs(n_mpi_procs))
displs(1) = 0
DO ii=1,n_mpi_procs-1
  displs(ii+1) = displs(ii) + nnode_owned_vector(ii)
END DO

CALL MPI_Allgatherv(nodelist_owned, nnode_owned, MPI_INT,
      node_num_get_old, nnode_owned_vector, displs, MPI_INT,
      PETSC_COMM_WORLD, errpetsc)

DO ii=1,nnode_global
  node_num_get_new(node_num_get_old(ii)) = ii
END DO
```

Remark 3: In general, the size of **nodelist_owned** array can be different on different processors. So, using **MPI_Allgather** leads to errors. Here, **MPI_Allgatherv** *must be used.*

Step2c: Update node numbers and generate data arrays

Once the nodes are renumbered, we can generate data arrays for creating the matrix pattern, assembling the element matrices and vectors, applying boundary conditions and post-processing.

- Update the node numbers for **elem_node_conn** array with new node numbers.

```
DO ee=1, nelem_global
  DO ii=1, nodes_per_elem
    n1 = elem_node_conn(ee,ii)
    elem_node_conn(ee,ii) = node_num_get_new(n1)
  END DO
END DO
```

- Update the node numbers for **dirichlet_data** array and create **dof_if_prescribed** and **soln_prescribred** arrays.

```
ALLOCATE( dof_if_prescribed(nnode_global,ndof_per_node) )
dof_if_prescribed=0

DO ii=1, nDBC
  n1 = node_num_get_new(dirichlet_data(ii,1))
  n2 = dirichlet_data(ii,2)

  dirichlet_data(ii,1) = n1

  dof_if_prescribed(n1,n2) = 1

  ind = (n1-1)*ndof_per_node + n2
  soln_prescribed(ind) = dirichlet_data(ii,3)
END DO
```

- Create **dof_to_sparsemtx**

```
ind = 1
DO ii=1, nnode_global
  DO jj=1,ndof_per_node
    IF(dof_if_prescribed(ii,jj) == 0) THEN
      dof_to_sparsemtx(ii,jj) = ind
      ind = ind + 1
    END IF
  END DO
END DO
ind = ind-1
```

- Compute first and last row indices, **row_start** and **row_end**, owned by the local processor.

```
row_start =  1e9
row_end   = -1e9
mat_size_local = 1
DO ii=node_start, node_end
  DO jj=1,ndof_per_node
    IF(dof_if_prescribed(ii,jj) == 0) THEN
      ind = dof_to_sparsemtx(ii,jj)
      row_start = MIN(row_start, ind)
      row_end = MAX(row_end, ind)
      mat_size_local = mat_size_local + 1
    END IF
  END DO
END DO
mat_size_local = mat_size_local - 1
```

- Generate **elem_assy_array**

```
DO ee=1,nelem_global
  DO ii=1,nodes_per_elem
    n1 = ndof_per_node*(ii-1)
    n2 = elem_node_conn(ee,ii)

    DO jj=1,ndof_per_node
      elem_assy_array(ee, n1+jj) = dof_to_sparsemtx(n2,jj)-1
    END DO
  END DO
END DO
```

- Compute **solver_to_mesh** array, the array to map the matrix solution to the global mesh.

```
ind = 1
DO ii=1,nnode_global
  DO jj=1,ndof_per_node
    IF(dof_if_prescribed(ii,jj) /= 0) THEN
      solver_to_mesh(ind) = (ii-1)*ndof_per_node + jj;
      ind = ind + 1
    END IF
  END DO
END DO
```

Step 2d: Create the matrix pattern

At this stage, all the required data arrays have been generated successfully. Now, we can generate the matrix pattern.

For the efficient assembly of the sparse matrix, knowing the exact pattern of the sparse matrix would be ideal. However, computing the exact matrix pattern requires the knowledge of all non-zero entries in the sparse matrix and this, in general, is difficult to compute as it requires arrays with dynamic memory allocation. While C++ provides efficient dynamic memory allocation for multidimensional arrays with arrays of various size in each row, it is difficult to do so in Fortran. So, in this work, we use a conservative estimate for the number of nonzeros in each row.

- Compute number of nonzeros in each row

```
n1 = 100; n2 = 50
IF(size_local < 50) THEN
  n1 = size_local
  n2 = n1
END IF

ALLOCATE(diag_nnz(n1))
ALLOCATE(offdiag_nnz(n2))

diag_nnz = n1
offdiag_nnz = n2
```

- Initialise the Petsc solver

```
reac_vec_size_local  = nnode_owned*ndof_per_node
reac_vec_size_global = nnode_global*ndof_per_node
call solverpetsc%initialise(
     reac_vec_size_local, reac_vec_size_global,
     mat_size_local, mat_size_global,
     diag_nnz, offdiag_nnz)
```

- Insert the non-zero entries into the sparse matrix

```
nsize = nodes_per_elem*ndof_per_node
Klocal = 0.0

LoopElem: DO ee=1, nelem_global
  IF(elem_proc_id(ee) == this_mpi_proc) THEN

    forAssyVec = elem_node_conn(ee,:)

    call MatSetValues(solverpetsc%mtx,
          nsize, forAssyVec, nsize, forAssyVec,
          Klocal, INSERT_VALUES, errpetsc)
  END IF
END DO LoopElem
```

Step 3: Calculate the global stiffness matrix and the load vector

- Initialise the global matrix and RHS vector to zero
- Loop over the elements and compute element stiffness matrix vector and residue vector, including body forces and applied Dirichlet boundary conditions

```
DO ee=1,nelem_global
  IF(elem_proc_id(ee) == this_mpi_proc) THEN
    *) evaluate element matrix & vector
    *) update element vector by applying the Dirichlet BCs
    *) assemble element matrix & vector
  END IF
END DO
```

Step 4: Solve the matrix system and update the solution vector for the mesh

- **soln_mesh** — type = double array
  - Stores the solution at all the nodes in the global mesh using the original node numbers
  - So, node mapping arrays must be used while post-processing

```
! solve the matrix system
solverpetsc%factoriseAndSolve()

! Store specified Dirichlet BC values
soln_mesh = soln_prescribed

! Add solution for the free DOF
! Scatter the matrix solution vector
! and get a pointer to the solution vector
call VecGetArray(vec_SEQ, xx_v, xx_i, errpetsc)

! for single dof problem
IF(ndof_per_node == 1) THEN
  DO ii=1,size_global
    ind = node_map_get_old(solver_to_mesh(ii))
    soln_mesh(ind) = xx_v(xx_i+ii)
  END DO
ELSE
! for multiple dof problems
  DO ii=1,size_global
    fact = xx_v(xx_i+ii)
    ind  = solver_to_mesh(ii)

    n1 = (ind/ndof_per_node)+1
    n2 = MOD(ind,ndof_per_node)
    IF(n2 == 0) THEN
      n1 = n1-1; n2 = ndof_per_node
    END IF

    kk = (node_map_get_old(n1)-1)*ndof_per_node+n2
    soln_mesh(kk) = fact
  END DO
END IF
```

Step 5: Post-process the results

Once the solution is computed, derived quantities for post-processing, for example stains, stress, energies etc., can be computed. These tasks are left to the reader. In the source code provided, the mesh together with element subdomain ids and the nodal solution are exported as a VTK legacy (.vtk) file which can be viewed with ParaView.

4. Test cases

In this section we discuss the parallelisation procedure using two two-dimensional test cases: a) Poisson equation in 2D and b) Linear elasticity in 2D.

4.1. Test case A: Poisson equation in 2D

We illustrate the arrays computed in the parallelisation routine using the sample 3×3 mesh shown in Fig. 2.

node no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
proc	0	0	0	0	1	0	0	0	1	1	1	0	1	1	1	1

Table 1: Test case A: **node_proc_id** array.

Proc 0	1,2,3,4,6,7,8,12
Proc 1	5,9,10,11,13,14,15,16

Table 2: Test case A: **nodelist_owned** array.

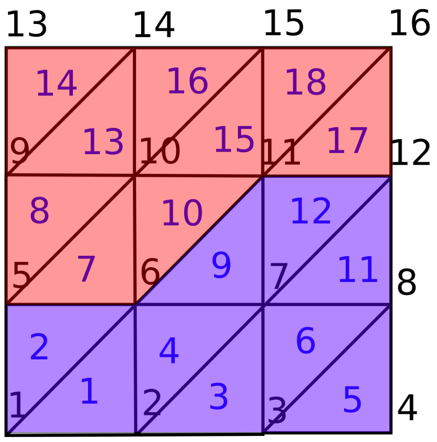

Figure 2: Sample mesh used for the 2D Poisson equation.

4.2. Test case B: Linear elasticity in 2D

5. Scalability studies

In this section we perform scalability towards assessing the performance of the parallelisation procedure implemented in this work. For this purpose, we consider 3D Poisson equation, 3D linear elasticity and 3D incompressible Navier-Stokes. For the sake of simplicity only static problems are considered.

new node num	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
old node num	1	2	3	4	6	7	8	12	5	9	10	11	13	14	15	16

Table 3: Test case A: **node_get_old** array.

old node num	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
new node num	1	2	3	4	9	5	6	7	10	11	12	8	13	14	15	16

Table 4: Test case A: **node_get_new** array.

node no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
val	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	1

Table 5: Test case A: **dof_if_applied** array. Here, node numbers after re-ordering, i.e. new node numbers.

node no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
val	0	0	0	0	1	2	0	0	0	0	3	4	0	0	0	0

Table 6: Test case A: **dof_to_sparse** array. Here, node numbers after re-ordering, i.e. new node numbers.

node no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ux	1	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0
uy	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Table 7: Test case B: **dof_if_applied** array. Here, node numbers after re-ordering, i.e. new node numbers.

node no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ux	0	1	2	3	4	6	8	10	0	0	14	16	0	19	21	23
uy	0	0	0	0	5	7	9	11	12	13	15	17	18	20	22	24

Table 8: Test case B: **dof_to_sparse** array. Here, node numbers after re-ordering, i.e. new node numbers.

nproc	Iters	Assy time (s)	Solver time (s)	Speed up - assy	Speed up - solver
1	634	26.46	275.80	1.0	1.0
5	864	4.01	62.19	6.6	4.4
10	938	2.00	37.86	13.2	7.3
20	1016	1.02	21.48	25.9	12.8
40	1065	0.52	11.93	50.9	23.1
80	1155	0.27	8.45	98.0	32.6

Table 9: Scalability study for 3D elasticity beam with 50*300*50*6 = 4,500,000 elements. PETSc options: -ksp_type cg; -pc_type bjacobi; -ksp_rtol 1.e-6; -ksp_max_it 2000; -sub_pc_type ilu; -sub_pc_factor_levels 1; -sub_pc_factor_fill 1; -sub_ksp_rtol 1.e-6

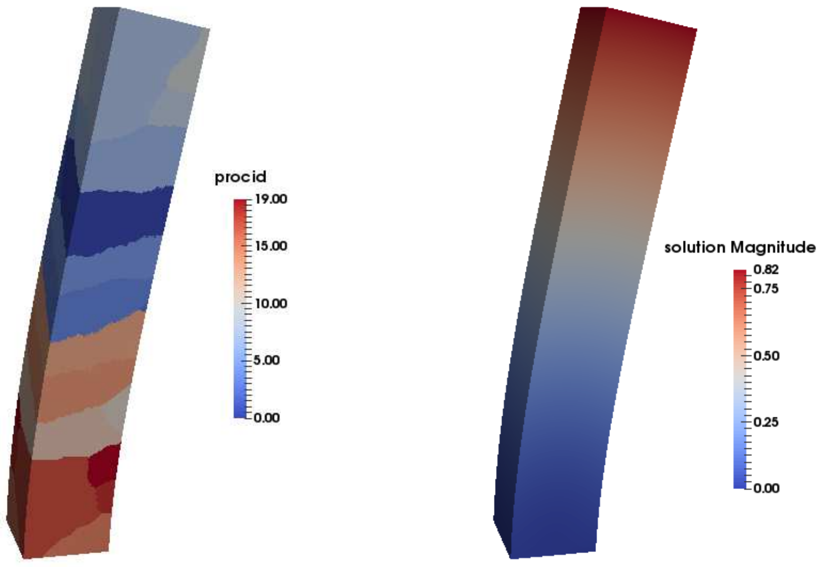

Figure 3: Linear elasticity in 3D: (a) Metis partitioning with 20 parts and (b) displacement magnitude obtained. (Need to create new eps files.)

no. of processors	Time (s)		Speed up	
	Stiff+Assembly	Solver	Stiff+Assembly	Solver
1	29.9	86.73	1.0	1.0
5	8.55	21.99	3.7	3.9
10	2.27	11.27	13.2	7.7
20	1.15	5.05	26.0	17.2
40	0.61	2.48	49.1	35.0
80	0.56	2.22	53.4	39.1

Table 10: Scalability study for 3D Poisson equation with $200 \times 200 \times 200 \times 6 = 48,000,000$ elements. Petsc options: -ksp_type cg -pc_type bjacobi, -ksp_rtol 1.e-6, -ksp_max_it 2000, -sub_pc_type ilu, -sub_pc_factor_levels 1, -sub_pc_factor_fill 1, -sub_ksp_rtol 1.e-6.

no. of processors	Time (s)		Speed up	
	Stiff+Assembly	Solver	Stiff+Assembly	Solver
1	26.46	275.80	1.0	1.0
5	4.01	62.19	6.6	4.4
10	2.00	37.86	13.2	7.3
20	1.02	21.48	25.9	12.8
40	0.52	11.93	50.9	23.1
80	0.27	8.45	98.0	32.6

Table 11: Scalability study for 3D linear elasticity; beam with $50 \times 300 \times 50 \times 6 = 4,500,000$ elements. Petsc options: -ksp_type cg -pc_type bjacobi, -ksp_rtol 1.e-6, -ksp_max_it 2000, -sub_pc_type ilu, -sub_pc_factor_levels 1, -sub_pc_factor_fill 1, -sub_ksp_rtol 1.e-6.