Chair for Computational Analysis of Technical Systems

RWTH AACHEN UNIVERSITY

Parallel Computing for Computational Mechanics

Summer semester 2021

Homework 4: Distributed Memory Parallelism with MPI

---

1 Introduction

In this Homework, you will parallelize the finite element code built in Homework 2 with the message passing interface (MPI) standard. The main idea is to distribute the data (elements and nodes) among the available processors (PEs).

2 Task Description

To get the code MPI ready, you will have to insert code in the `tri.cpp` and `solver.cpp` files. Complete the following tasks:

- Task 1: In the `readMeshFiles()` routine in `tri.cpp`, determine the number of elements on the current PE (**nec**), the maximum number of elements across all PEs (**mec**), the number of nodes per PE (**nnc**), and the maximum number of nodes across all PEs (**mnc**). For this task, make sure to distribute the nodes and elements as evenly as possible amongst all PEs. You can use the algorithm provided in the lecture slides.
- Task 2: In `localizeNodeCoordinates()` implement the localization of coordinate data. Localization means that the local node structure `lNode` contains the x and y coordinates of the nodes on each PE from the global node list. Each PE has its own set of partitioned xyz coordinates. You have to create an MPI window containing the coordinate vector `xyz` per PE and go over all the nodes on the respective PE and set the global coordinates through the `setX` and `setY` functions. For this purpose use to mapping function `nodeLToG`. For the localization, use `MPI_Get()`.
- Task 3: In `solver.cpp` complete the routine `accumulateMass`. In this routine, rewrite the global mass per partition structure to a local mass per partition structure. The key part of this routine is to figure out how to determine whether the current node lies on the current PE and if so, how far the offset of the node on the PEs. With this information, you can then map the mass value with a global node number to the buffer with the local node number. Via the `MPI_Accumulate` function, you make sure that entries are added up and not overwritten.

- Task 4: Complete the function `localizeTemperature`. As the name states, this is also a localization routine. It is used to get the new temperature value of `TG`. This is an array that lives on each PE, but with a global node numbering. As we want to have the nodes in a local numbering, you once again use `MPI_GET()`. Go over all the nodes, and if one node belongs to the current PE, determine the offset. With the offset, provide the mapping from the buffer (which is `TG`) to the local temperature array per partition `TL`.
- Task 5: Similar to Task 3, now accumulate the data on the `MTNew` vector in `accumulateMTNew`. This function follows the same structure as Task 3.
- Task 6: Add the calls for the routines you completed in Tasks 1-5 to the `solver.cpp`, whenever there is a comment that says `ADD MPI COMMUNICATION HERE`.

3 Hints for the Tasks

3.1 Global and Local Variables and Data

Compared to OpenMP, parallelizing a code with MPI is more intrusive. To distribute the data amongst multiple PEs, the data has to be split adequately. If we consider this in the context of solving the heat equation on a given domain, each PE now takes care of a partition of the domain. Wherever data overlaps betweens PEs (e.g. nodes that are part of multiple mesh partitions), it is communicated using MPI directives. Therefore, one uses the expressions *global* and *local* for data structures. *Global* refers to everything that is valid for the entire domain and for a serial code (using only 1 PE as in Homework 2), *local* refers to all the data that lives on a certain PE.

To distribute the global data among PEs, three different arrays are used, i.e. local element arrays, local node arrays, and local element node arrays with dimension `nec`, `nnc`, and `nnl` respectively.

- **Local element arras:** this array contains a portion of the elements of the computational mesh which are part of the current PE. Hence, this array is of size `nec`.
- **Local node array:** this contains a portion of the nodes of the computational mesh which are part of the current PE. Note that these node do not have to correspond with the nodes of the elements in the local element array. The local node arrays are of size `nnc` and contains the global node numbering.
- **Associated element nodes array:** this arrays contains the nodes that are associated with the elements in the local element array. Hence this array is of size `nnl`.

An example of array distribution for a simple mesh over 4 PEs is presented in Figure 1.

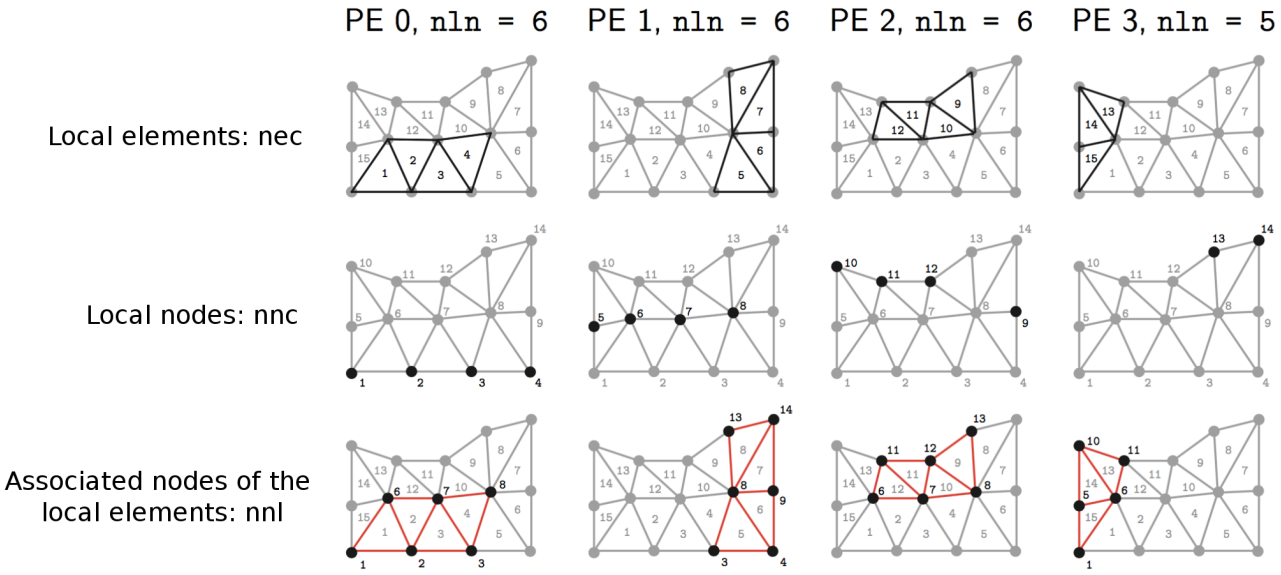

Figure 1: Example element and node data distribution (`ne = 15, nn = 14, npes = 4`).

For example, on PE 0, the data related to elements 1, 2, 3, and 4 (`nec`) are stored alongside the data related to nodes 1, 2, 3, and 4 (`nnc`).

- Arrays indexed with `nec` contain data of local elements (stored on the current PE). Example: `mesh->getElem(nec)`
- Arrays indexed with `nnlc` contain data of local nodes (stored on the current PE). Example `TG[nnc]`[1].
- Arrays indexed with `nnl` contain data of the nodes of the local elements. Example: `TL[nnl]`

Some examples of node arrays corresponding the to global mesh and the local elements is given in Table 1

Table 1: Global and local arrays.

Description	Global	Local
Temperature	`TG[nnc]`	`TL[nnl]`
Right hand side	`MTnewG[nnc]`	`MTnewL[nnl]`
Mass vector	`massG[nnc]`	`mesh->getLNode(nnl)->getMass() (class)`
Node coordinates	`xyz[nsd,nnc]`	`mesh->getLNode(nnl)->getX()/getY() (class)`

Your job is to correctly distribute the global data to the PEs. With the data distributed among the PEs, the solver parallelized with MPI works similar to the serial solver.
The mapping from local to global level done in the `formLocalNodeList()` routine in `tri.cpp`. As can be seen in in Figure 1, the local nodes and associated nodes do not need to coincide. Hence, in order to construct the associated element nodes array that corresponds to the elements on the current PE, nodal

[1]Notice that, although this array is local (on current PE), it contains global node numbers. Hence the use of `G` in the name `TG[nnc]`

data from other PEs needs to be retreived. Figure 2 and Figure 3 outline the steps to obtain the correct mapping. These steps can be summarized as given below.

1. Store the global node numbers occuring in the locale element connectivity in `allLocalNodes`.
2. Sort `allLocalNodes` and keep track of the indexing.
3. Obtain the number of associated element nodes `nnl` by counting the number of unique node numbers in `allLocalNodes`. Assign a new (local) index to each unique node number in `allLocalNodes` and store these in `rawLocalConn`.
4. Store the unique global nodes occurring in `allLocalNodes` inside `nodeLToG`.
5. Using the `nodeLToG` and initial `allLocalNodes`, array, retrieve the local nodes and store them in `allLocalNodes`.

After the last step, `allLocalNodes` is set to the element object by the `setConn` function. The example in Figure 2 and Figure 3 is shown in the comments of `tri.cpp`.

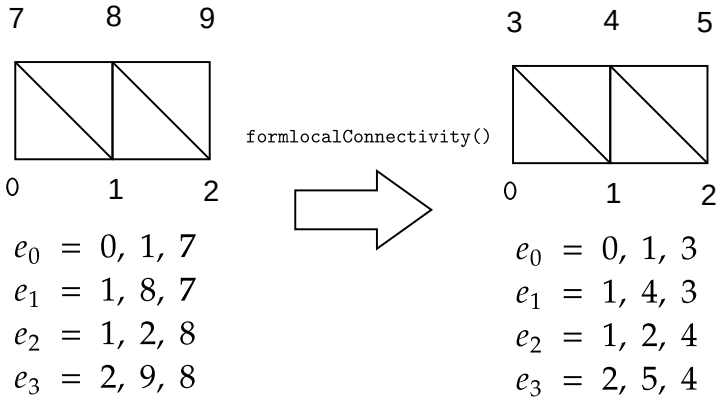

Figure 2: Example mesh with 6 nodes and 4 elements with global (left) and local(right) node numbering.

```
Step 1:
allLocalNodes = 0,1,7,1,8,7,1,2,8,2,9,8
index         = 0,1,2,3,4,5,6,7,8,9,10,11

Step 2: sorts
allLocalNodes = 0,1,1,1,2,2,7,7,8,8,8, 9
index         = 0,1,3,6,7,9,2,5,4,8,11,10

Step 3:
nnl           = 6
rawLocalConn  = 0,1,1,1,2,2,3,3,4,4,4,5

Step 4:
nodeLToG      = 0,1,2,7,8,9

Step 5:
allLocalNodes = 0,1,3,1,4,3,1,2,4,2,5,4
```

Figure 3: Behavior of the `formLocalNodeList()` algorithm.

3.2 MPI Routines

From the tasks in section 2, you see that two main ways of MPI one-sided communication are used, localization and accumulating. Localization, is simply getting information across PEs. Accumulating includes the addition of values from multiple PEs onto a single PE. The routines you will use here are called `MPI_Get()` and `MPI_Accumulate()`.
Before you start the actual communication, a window has to be opened via `MPI_win_create()`. To make sure that all processes wait and synchronize, you should fence the window with `MPI_Win_fence()`. Further important MPI routines are `MPI_Comm_Size()` that returns the number of PEs. Also, `MPI_Barrier()` is used to synchronize PEs at the end of all MPI communication.

4 Validating Your Work

Run yout parallelized MPI code for the coarse, fine and finest mesh with the respective settings files. To run a job interactively (not recommended), for a example on 2 cores, you could use the following command from the test directory:

```
mpiexec -n 2 ../build/2d_Unsteady settings.coarse.in
```

We rather recommend to use the provided `run.j` batch script. To change the number of cores, put in the desired number of cores in `#SBATCH -ntasks=< number of cores>` flag. Run the meshes for 1, 2, 4, 6, 8, 12, 16 cores. Compare the execution time. Calculate the parallel efficiency and speed-up. In addition you

should consider the following remarks:

- Make sure that the converged temperature field is the same irrespective of the number of cores.
- Make your program general enough to run unaltered on any number of PEs.
- What would be a better MPI communication strategy for our problem? Why?
- What is the advantage of one-sided communication?
- How would you improve the code?

IMPORTANT: The exclusive flag, `"#SBATCH --exclusive"`, is meant to be used to collect timings, but not for testing or debugging. Make sure it is set correctly when collecting runtime data. Please use computing resources conscientiously. Computations on large numbers of CPUs should performed when you are certain your code is working correctly.

Contact:
Max Schuster · schuster@cats.rwth-aachen.de
Michel Make · make@cats.rwth-aachen.de