Chair for Computational Analysis of Technical Systems

RWTH AACHEN UNIVERSITY

Parallel Computing for Computational Mechanics

Summer semester 2020

Project 3

In this project, you will parallelize the finite element code considered in Projects 1 and 2 with MPI. The main idea is to distribute the data (elements and nodes) among the available PEs. It allows computation to be done independently on each PE and transferred to and from other PEs when needed.

Two types of data will be distributed: element and node data. An example of data distribution can be found hereafter:

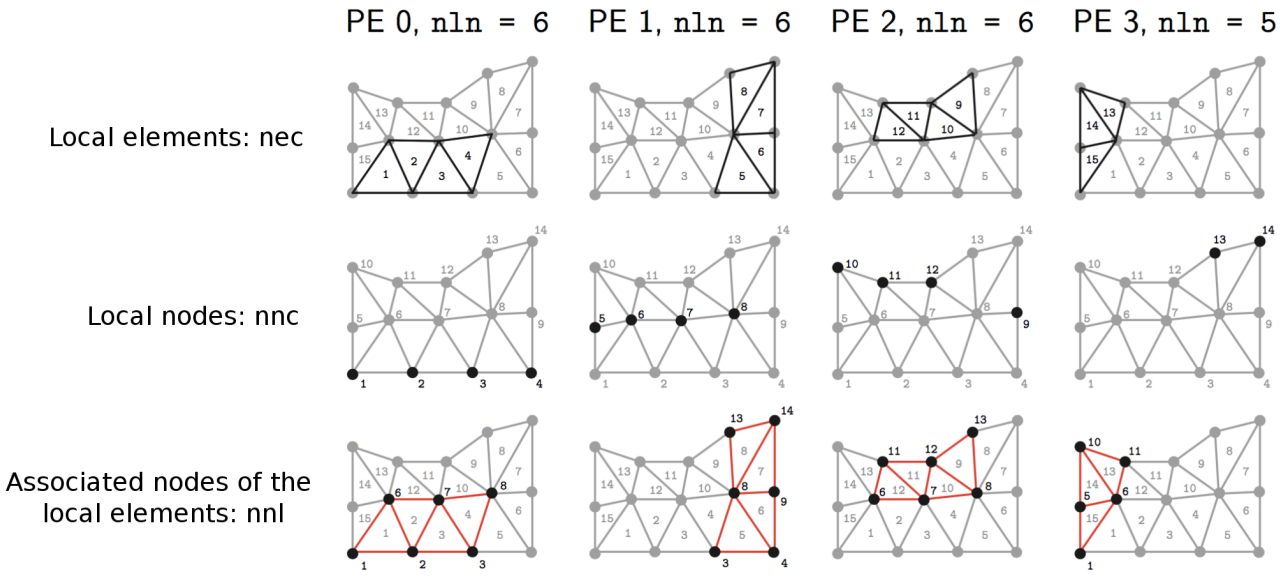

Figure 1: Example element and node data distribution (`ne = 15, nn = 14, npes = 4`).

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

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)

Table 1: Global and local arrays

Your will have to add code to the `tri.cpp` and `solver.cpp` files.

- In `tri.cpp`, complete the following functions:
  1. Inside `readMeshFiles()`, you first have to partition the elements and nodes. In other words, you have to find:
     (a) `nec`: number of elements stored on the current PE.
     (b) `mec`: maximum number of elements across all PEs.
     (c) `nnc`: number of nodes stored on the current PE.
     (d) `mnc`: maximum number of nodes across all PEs.
  2. Inside `localizeNodeCoordinates()`, you have to implement the localization of coordinate data (from `xyz[nnc]` to `lnode[nnl].setx/.sety`).
- In `solver.cpp`, complete the following functions:
  1. `accumulateMass`: You have to transfer data from the node class (`mesh->getLNode(nnl)->getMass()`) to `massG[nnc]`). **Why are we using using `MPI_Accumulate` and not `MPI_Put`** (Please answer this question in your report)?
  2. `localizeTemperature`: You have to transfer data from `TG[nnc]` to `TL[nnl]`.
  3. `accumulateMTnew`: You have to transfer data from `MTnewL[nnl]` to `MTNewG[nnc]`.
- Replace the comments "`Add MPI Communication here`" by the appropriate communication routine.
- Finally, check that the the code is producing correct result by looking into the deviation from the analytical solution.

For the temperature localization, a pseudo-code could look similar to this:

```
initialize buffer of length mnc to zero
loop over pes
        determine nncTarget
        MPI_Get(&buffer[0], nncTarget, MPI_DOUBLE, ipes, 0, nncTarget, ...
        ... MPI_DOUBLE, win);
        some more MPI communication

        loop over inl
                determine position and offset of inl on global level
                assign the respective value from the buffer to the ...
                ... local temperature array for the  appropriate PE

some more MPI communication
delete[] buffer
```

You can find the main helper functions hereafter:

- `mesh->getNodeLToG()`: return the pointer to the array `nodeLToG[nnl]` which stores the global node number (`nn`) corresponding to a local node number (`nnl`).
- `mesh->getMnc()`: return maximum number of nodes across all PEs.
- `mesh->getNnl()`: return maximum number of nodes of the elements stored on the current PEs.
- `mesh->getNnc()`: return number of nodes stored on the current PE.
- `mesh->getMassG()`: return pointer to the `massG[nnc]` array.
- `mesh->getLNode(i)->getMass()`: return mass of the local node `i` ($0 \leq i < nnl$).
- `mesh->getMTnewG()`: return pointer to the `MTNewG[nnc]` array.
- `mesh->getMTnewL()`: return pointer to the `MTnewL[nnl]` array.
- `mesh->getTG()`: return pointer to the `TG[nnc]` array.
- `mesh->getTL()`: return pointer to the `TL[nnl]` array.

In addition you should consider the following remarks and answer the question:

- 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.
- **How would you improve the code?**

Trace Information with Vampir for 2 Bonus points

Finish this section to gather up to 2 bonus points. In order to visualize better the communication pattern, you will instrument the code to gather trace information using SCALASCA and VAMPIR. You need to have logged in into the cluster with X-Forwarding turned on, e.g., by using `ssh` with the `-Y` flag (or with FastX). You can load the necessary modules via:

```
module load DEV-TOOLS scalasca scorep vampir
```

During the compilation step, "scorep" has be placed before the compiler command. It can be done by changing the following environment variable:

```
export CXX="scorep icpc"
export SCOREP_TOTAL_MEMORY=475MB
```

Trace analysis has significant overhead. When you want to generate a trace, reduce the max number of iteration inside "settings.fine.in": "iter 10" for example. The trace of the program can be then obtained by submitting a job using the provided run.sh script. The results of the tracing is placed into a sub-folder named:

```
./scorep_2d_Unsteady_<n>xP_trace/
```

Where <n> is the number of PEs. The communication pattern is finally obtained by running:

```
vampir ./scorep_2d_Unsteady_<n>xP_trace/traces.otf2
```

Use color coding to highlight the different steps of the communication and computation. Further information can be found in the primer of the RWTH HPC-Cluster (Link). It shall also be noted that it might be necessary to zoom into the communication pattern plot, which can be easily accomplished by clicking into the plot and selecting a range.

Submission Guidelines

Turn in a report (pdf file, ideally containing your last name in the file name) containing

- excerpts of essential parts of the code and **explanations in the report** for each code section you implemented (`readMeshFiles()`, `localizeNodeCoordinates()`, `accumulateMass()`, `localizeTemperature()`, `accumulateMTNew()`),
- a (line) plot comparing your parallelized solution to the analytical or serial solution for verification,
- a (line) plot to prove that the solution stays unaltered for any number of cores (compare for at least 3 different numbers),
- timings and efficiency analysis of your code on $1, 2, 4, 8, 16, 32$ cores (if possible, use more cores) (please use the `#SBATCH -exclusive` option only for final timings). Plot the speed-up and efficiency over the number of cores and comment on the outcome, **Update: Please, obtain the timings for all 3 meshes (coarse, fine, finest). Plot the efficiency and speed-up for all 3 meshes and comment on the differences in the behavior**,
- a sketch or Vampir-generated graphics of the communication pattern for one iteration of the solver with a explanation and interpretation (`nIter=1`) (optional for 2 bonus points),
- a self written introduction explaining the scope of the project and a structural summary,
- answers to the 2 questions from the text, 1) about why we are using `MPI_Accumulate` over `MPI_Put` and 2) how to improve the code further,

- a conclusion, wrapping up the project and what you found out in this assignment.
- We provide slightly different meshes now, where coarse is still the old coarse mesh, but finest is now the fine mesh and the finest one is even finer than usual. Please use coarse for code development, but you can use fine for the production run where you compare the analytical/serial solution to the parallel one. For the finest, the number of iterations is lower than it would need to fully converge, but you can still use this mesh for parallelization efficiency/speed-up analysis.

In addition also provide

- the self-contained source code as 1 instance in a top-level folder called "Code" (must compile and run on the RWTH ITC Cluster).

All this should be wrapped within a single tar file named `pr3.tar.gz`, which can be, e.g., created by the command:

```
tar czvf pr3.tar.gz pr3/*
```

Please do not use other archive formats. Furthermore, the following guidelines apply:

- Solutions are accepted until July 19, 11:55 PM.
- The solution to this project shall be done individually (not as a team) and handed in online by using the $RWTHmoodle$ system.
- A file `pr3.tar.gz` containing the data and the skeleton of the program can be found on the $RWTHmoodle$ page.
- Do not add additional source files.
- The code must compile and run on the RWTH ITC Cluster.

Contact:
Maximilian Schuster · schuster@cats.rwth-aachen.de
Violeta Karyofylli · karyofylli@cats.rwth-aachen.de