High Performance Computing (Master) in WS19

Exercise 4: distributed memory parallelization using MPI

Deadline: Feb 10, 2020, 16:00

Status: Thursday 23rd January, 2020, 08:58

Supervisor: Prof.Dr. G. Haase, gundolf.haase@uni-graz.at

1 Preliminaries

1.1 MPI

The **M**essage **P**assing **I**nterface has been introduced in the early 90–ties (i.e., after Rocky V) and it is still the standard environment for distributed parallel computing. It covers about 140 functions, available in F77, C and C++. Already 6 functions allow to write parallel codes. Most of the other functions are based on these 6. We will be mainly concerned with the following functions.

Basic functions	**MPI_Init**
	MPI_Finalize
	MPI_Send
	MPI_Recv
	MPI_Comm_rank
	MPI_Comm_size
additional functions	**MPI_Barrier**
	MPI_Bcast
	MPI_Gather
	MPI_Scatter
	MPI_Reduce
	MPI_Allreduce

The current standard is MPI-3.1 although this basic course will use only a small subset of its funtionality.

1.2 Online help

First of all the MPI Homepage and especially the overview of the MPI functions should be consulted. We will refer frequently to these web pages during the course.

The desciption for MPI–functions in C and Fortran (nad Fortran 2008) can be found **here**. The c++ bindings were deprecated.

We have to distinguish between the MPI standard and its implementations. The most commenly used implementations are MPICH and OpenMPI (that is not OpenMP !!). All three are available as packages under LINUX but only one of them should be used in order to avoid confusion wrt. paths to executables, libraries and headers. We will refer to the latter one, see the man pages of OpenMPI.

1.3 Getting started on a (pool of) LINUX-workstations/PCs

First, open a shell and type

```
mpirun
```

If MPI is not available then you have to install additional packages (in Ubuntu) via

```
sudo apt-get install openmpi-bin openmpi-doc libopenmpi-dev
```

or install it from the scrach (just for fun).

Check whether the ssh-deamon is running

```
ps -ax |grep sshd
```

If not you have to install it too.
In order to avoid the password request for each parallel process started (think of 64 parallel processes) you have to create secure authentication keys for your account.

```
ssh-keygen
cp  ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys
chmod go-rwx ~/.ssh/authorized_keys
# locally
ssh-add
# remote
ssh-copy-id your_username@192.0.2.0
```

See also my hints.

1.4 Installing the example code

Copy and unpack the provided supporting material for the Jacobi template into a folder and unpack it.

2 Your first parallel code

E1 Compile the program in first.template and set in the *Makefile* the variable `COMPILER` to `COMPILER=GCC_` . Adapt *Makefile* and *../GCC_default.mk* to your needs and paths. Compile and link the code (1 pts)

```
make
```

Start the program with 4 processes

```
make run
```

or directly via

```
mpirun -np 4 ./main.GCC_
```

(ensure that you use the right `mpirun`)

If you 'mpirun ...' report some error "... not enough slots .." then use the option '–oversubscribe', i.e., `mpirun --oversubscribe -np 4 ./main.GCC_`

The following MPI functions require a *communicator* as parameter. This communicator describes the *group* of processes which are to be covered by the corresponding MPI function. By default, all processes are collected in *MPI_COMM_WORLD* which is one of the constants supplied by MPI for predefined datatypes, error classes, collective operations etc.. We restrict the examples to those global operations. For this purpose, create special MPI-type variable *MPI_Comm icomm= MPI_COMM_WORLD;* which is used as parameter !

E2 Write Your first parallel program by implementing (1 pts)

MPI_Init and **MPI_Finalize**

compile the program and start 4 processes

```
mpirun  -np 4  ./main.GCC_
```

E3 Determine the number of your parallel processes and the local process rank by using the routines (1 pts)

MPI_Comm_rank and **MPI_Comm_size** .

Let the root process (0==rank) write the number of running processes. Start with different numbers of processes.

E4 The file *greetings.cpp* includes a routine (1 pts)

Greetings(MPI_Comm const &icomm)

that prints the names of the hosts your processes are running on. Call that routine from your main program and change the routine such that the output is ordered with respect to the process ranks. Study the routines

MPI_Send and **MPI_Recv**

with respect to *tag*s and *rank*s.

3 Global Operations

Let some Double Precision vector $\underline{x}$ be stored blockwise disjoint, i.e., distributed over all processes s (s=0,...,P−1) such that $\underline{u} = (\underline{x}_0^T, \ldots, \underline{x}_{P-1}^T)^T$. See the suggestion for the function interface.

E5 Write a routine (1 pts)

DebugVector(xin, icomm)

that prints the Double Precision vector *xin*. Start the program with several processes. $\Longrightarrow$ All processes will write their local vectors, i.e., one has to look carefully for the data of process s.

Improve the routine **DebugVector** such that process 0 reads the number (from terminal) of that process which should write its data next. Use

MPI_Bcast

to broadcast this information and let the processes react appropriately. If necessary use **MPI_Barrier** to synchronize the output.

E6 Write a routine for computing the global scalar product (1 pts)

par_scalar(x, y, icomm)

of two Double Precision vectors x and y of local length n. Use

MPI_Allreduce with the operation **MPI_SUM**.

E7 Exchange global minimum and maximum of the vector $\underline{x}$! Use (1 pts)

MPI_Gather , **MPI_Scatter** , **MPI_Bcast** and **MPI_Sendrecv** .

How can you reduce the amount communication?
Hint: Compute, first, local min./max. and afterwards let some process determine the global quantities.

Alternatively, you can use **MPI_Allreduce** and the operations **MPI_Minloc**/**MPI_Maxloc**.

E8 Each of your 4 MPI processes (numprocs := 4) owns a double precision vector x with 20 elements initialized as (1 pts)

$$x[i] := \text{myrank} * 100 + (i \,\%\, 5) * 10 + i \ .$$

You have to exchange vector elements with MPI ranks $r \in [0, 3]$. All elements with index $i \in [r * 5, (r + 1) * 5 - 1]$ have to be exchanged with rank r.
Check the resulting vector.

You could use a bunch of **MPI_Sendrecv** calls but you better use the more general function **MPI_Alltoall** .
Try also the In-place option by using *MPI_IN_PLACE* with **MPI_Alltoall** .

Have a look at the more general function **MPI_Alltoallv** .

4 Data exchange

Download the template containing the functions for the setting up the geometry and the exchange of data between MPI processes.

For a start we use Matlab to describe manually the decomposition of a square into sub-squares, see script square_4.m. The resulting list of finite elements together with the node coordinates is stored in square_4.txt. The assignment of elements to a subdomain (= MPI rank) $r = 0, \ldots, 3$ is stored in file square_4_sd.txt.

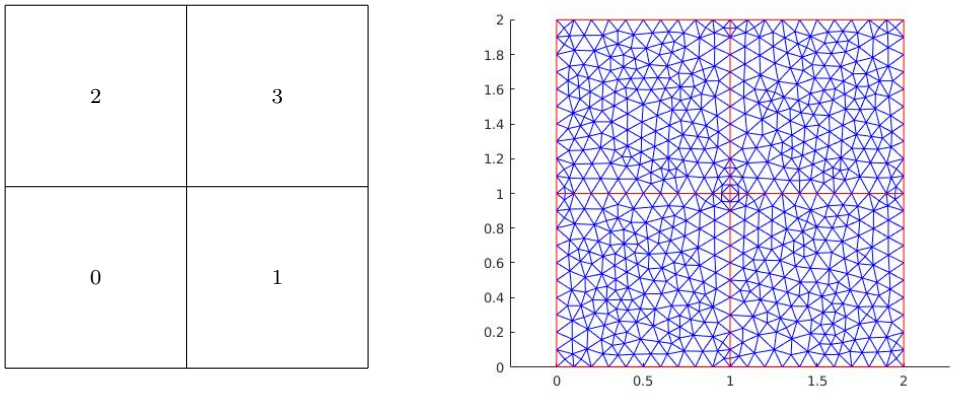

Figure 1: MPI rank per subdomain (left) and discretization (right).

Please note that both text files contain the original Matlab numbering starting with 1 which will be changed automatically to C-numbering (starting with 0) in the C++-constructor of the mesh. The class ParMesh reads these files with its constructor

ParMesh const mesh(”square”,icomm);

for the provided files (*only for 4 MPI processes!*) and initializes also the MPI parallelization for data exchange on the interfaces of the subdomains.

Object `mesh` contains basic communication routines and information as

- Number of MPI processes: `int const numprocs = mesh.NumProcs();`
- MPI rank of this process: `int const myrank = mesh.MyRank();`
- inner product: `double ss = mesh.dscapr(xl,xl);`
- Vector accumulation on interfaces: `mesh.VecAccu(xl);`

E9 Check the vector accumulation `mesh.VecAccu(...)` with your own vector taking into account that you know the interface coordinates. The coordinates can be accessed via `mesh.GetCoords();` . (2 pts)
See also Figure 2 for a scheme of the available local node numbering.
Check also the inner product `mesh.dscapr(...)` .

E10 Write a Method `VecAccu` for class `Parmesh` that adds interface data of an *integer* vector. (2 pts)

E11 Write a Method `GlobalNnodes()` for class `Parmesh` that determines the global number of nodes in the mesh. (2 pts)

E12 Write a Method `Average` for class `Parmesh` that averages (arithmetic mean) interface data of a double precision vector instead of adding them. (2 pts)

E13 Generate other domains for parallel data exchange by copying square_4.m to a different name and describe your subdomains in a similar way. (2 pts)
Try also more/less than 4 subdomains and start your code appropriatly, see target `run` in *Makefile*.

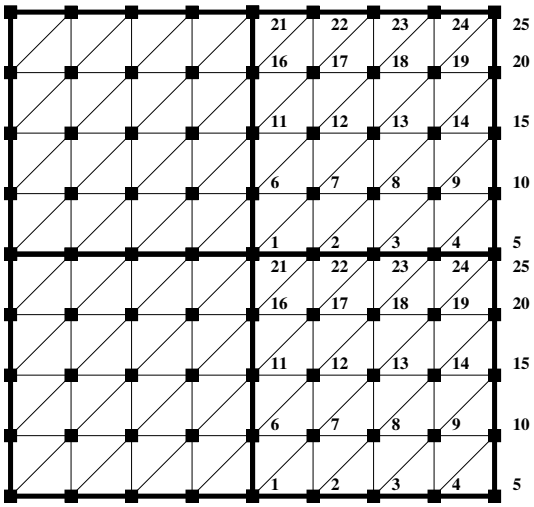

Figure 2: 4 subdomains in local numbering.

5 Iterative Solvers

Download the template containing the functions for matrix generation and sequential solvers or use your own version from the shared memory part of the course.

As model problem, we consider the homogeneous Dirichlet boundary value problem ($\mathbf{u}(x) = 0 \quad \forall x \in \partial\Omega$) for the Poisson equation in the unit square $\Omega := (0,1)^2$ in its weak formulation:

Find $\mathbf{u} \in \mathbf{H}_0^1(\Omega)$ such that

$$\int_\Omega \nabla^T \mathbf{u}(x) \nabla \mathbf{v}(x)\, dx \;=\; \int_\Omega \mathbf{f}(x)\mathbf{v}(x)\, dx \quad \forall\, \mathbf{v} \in \mathbf{H}_0^1(\Omega) \;. \tag{1}$$

We use linear finite elements for the discretization and achieve the linear system of equations

$$K \cdot \underline{u} \;=\; \underline{f} \;. \tag{2}$$

5.1 ω-Jacobi solver

Let us denote the diagonal of matrix K by $D = \mathrm{diag}(K)$. Now, we can formulate the ω-Jacobi iteration

$$\underline{u}^{k+1} \;=\; \underline{u}^k + \omega \cdot D^{-1} \cdot (\underline{f} - K \cdot \underline{u}^k) \quad , \quad k = 0, 1, 2, \ldots \;. \tag{3}$$

You will find a sequential version of the Jacobi solver in the directory *jacobi.template* extended with the parallel mesh and its functions provided already in the previous section 4.

Once you have an object `mesh` of class `Parmesh` the following matrix generation steps require **no communication**:

- Allocation and patter determination of local sparse matrix:
  `FEM_Matrix SK(mesh);`
- Calculation odf matrix entries (for Laplacian):
  `SK.CalculateLaplace(fv);`
- Setting values for local vectors, depending e.g. on node coordinates:
  `mesh.SetValues(...)`
- Incorporating boundary conditions (here, for Dirichlet):
  `SK.ApplyDirichletBC(...)`

The only function remaining for MPI adaption is `JacobiSolve(SK, fv, uv );`

E14 Implement an MPI parallel version of the sequential code in `JacobiSolve` with the interface (4 pts)

```
void JacobiSolve(ParMesh const & mesh, CRS_Matrix const &SK,
                 vector<double> const &f, vector<double> &u)
```

which requires the use of `mesh.dscapr(...)` and `mesh.VecAccu(...)`. The latter one is needed also in accumulation of the matrix diagonale.

This document will be extended by further advices, links, etc.

Thursday 23rd January, 2020