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

Although there exists an MPI–2 standard we restrict ourselfs in the beginning to the MPI–1 standard.

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 C/Fortran) can be found **here**.

We have to distinguish between the MPI standard and its implementations. The most commenly used implementations are MPICH, LAM 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 --no-install-recommends
```

or install it from the scrach (just for fun). The `--no-install-recommends` prevents the removal of slightly incompatible packages as CUDA 6.5. under ubuntu 14.10.

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
ssh-add
```

See also my hints.

1.4 Installing the example code

Copy and unpack the provided supporting material for C++
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

```
make
```

Start the program with 4 processes

```
mpirun -np 4 first.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. 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

MPI_Init and **MPI_Finalize**,

compile the program and start 4 processes

```
mpirun  -np 4  first.LINUX
```

E3 Determine the number of your parallel processes and the local process rank by using the routines

MPI_Comm_rank and **MPI_Comm_size**.

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

E4 The file *greetings.cpp* includes a routine

Greetings(const MPI::Intracomm& 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 Synchronized Communication

E5 Write a routine

Send_ProcD(to,nin,xin,icomm)

which sends *nin* Double Precision numbers of the array *xin* to the process *to*. Note that the receiving process *to* usually has no information about the length of the array it receives.
Also write a corresponding routine

Recv_ProcD(from,nout,xout,maxbuf,icomm)

which receives *nout* Double Precision numbers of the array *xout* from the process *from*. A priori, the receiving process does not have any information about the length of the data to be received, i.e., *nout* is an output parameter. *maxbuf* stands for the maximum length of the array *xout*.

E6 Test the routines from E5 with two processes first. Let process 1 send data and process 0 receive them. Extend the test to more processes.

E7 Combine the routines from E5 to one routine

ExchangeD(yourid,nin,xin,nout,xout,maxbuf,icomm),

which exchanges double precision data between the own process and another process *yourid*. The remaining parameters are the same as in E6. Test your routines with two and more processes.

E8 Implement a version of function **Exchange** that uses synchronous communication.

4 Global Operations

Let some Double Precision vector $\underline{u}$ be stored blockwise disjoint, i.e., distributed over all processes s (s=0,...,$P-1$) such that $\underline{u} = (\underline{u}_0^T, \ldots, \underline{u}_S^T)^T$.

E9 Write a routine

DebugD(nin,xin [,icomm])

that prints *nin* Double Precision numbers of the array *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 **DebugD** such that process 0 reads the number (from terminal) of that process which is to write it's vector. Use

MPI_Bcast

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

E10 Exchange global minimum and maximum of the vector $\underline{u}$! Use

MPI_Gather , **MPI_Scatter** / **MPI_Bcast** and **ExchangeD**.

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**.

E11 Write a routine for computing the global scalar product

Skalar(n,x,y [,icomm])

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

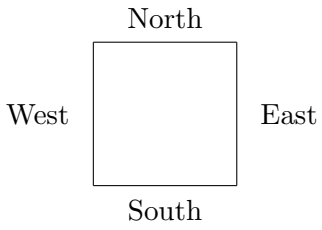
MPI_Allreduce with the operation **MPI_SUM**.

5 Local data exchange

Download the template containing the functions for the setting up the geometry.

Let the unit square $[0,1]^2$ be partitioned uniformly into $procx \times procy$ rectangles Ω_i numbered row by row. The numbering of the subdomains coincides with the corresponding process-id's (ranks).

$procx*(procy-1)$		$procx*procy-1$
$procx$		
0	1	$procx-1$



The function

IniGeom(myid,procx,procy,neigh,color)

generates the topological relations corresponding to the domain decomposition defined above. These information are stored in the integer array `neigh[4]`. A check–board coloring is defined in `color`. Moreover, the function

IniCoord(myid,procx,procy,xl,xr,yb,yt)

can be used to generate the coordinates of the lower left corner (xl, yb) and the upper right corner (xr, yt) of each subdomain.

E12 Realize a local data exchange of a double precision number between each processor and all of it's neighbors (connected by a common edge). Use the routine **ExchangeD** from E8.

Let each subdomain Ω_i be uniformly discretized into $nx * ny$ rectangles generating a triangular mesh (nx, ny are the same for all subdomains !) as depicted in Fig. 1

If we use linear f.e. test functions then each vertex the triangles represents one component of the solution vector, e.g., the temperatur in this point, and we have $nnode := (nx+1)*(ny+1)$ local unknowns within one subdomain. We propose a locally rowise ordering of the unknowns. Note, that the global number of unknowns is

$$N = (procx * nx + 1) * (procy * ny + 1) < procx * procy * nnode) \ .$$

The coordinates and the finite element mesh (triangular linear elements) are generated for each subdomain via:

GetMesh(nx, ny, xl, xr, yb, yt, nnode, xc, nelem, ia)

This function returns the number of nodes `nnode` and the number elements `nelem` together with the allocated and initialized coordinate vector `xc[nnode*2]` and the element connectivity `ia[nelem*3]`.

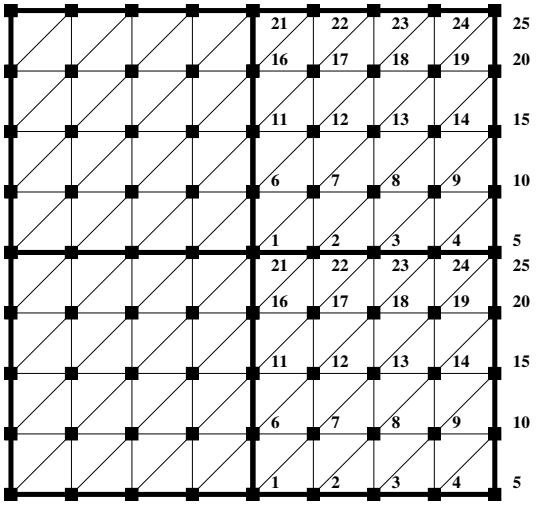


Figure 1: 4 subdomains in local numbering with local discretization $nx = ny = 4$ and global discretization $N_x = N_y = 8$.

The function **GetBound(ib,nx,ny,w,s)**

copies the values of w corresponding to the boundary South(ib=1), East (ib=2), North (ib=3), West (ib=4) into the auxiliary vector s. Vice versa, the function

AddBound(ib,nx,ny,w,s)

adds the values of s to the components of w corresponding to the nodes on the boundary South(ib=1), East (ib=2), North (ib=3), West (ib=4). These functions can be used for the accumulation (summation) of values corresponding to the nodes on the interfaces between two adjacent subdomains which is a typical and necessary operation.

E13 Write a routine which accumulates a distributed Double Precision vector w. The call of such a routine could look as follows

VecAccu(nx,ny,w,neigh,color,icomm)

where w is both in- and output vector.

6 Iterative Solvers

Download the template containing the functions for matrix generation and sequential solvers.

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

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

6.1 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* with the following functions in addition to the functions from P 5.

The function **Get_Matrix_Pattern(nelem, 3, ia, nnz, id, ik, sk)**

generates the pattern (`id[nnode+1]`, `ik[nnz]`) for a sparse matrix stored in CSR format with `ik[j]` as index where row j starts. The number of non-zero elements `nnz` is determined from the given discretization with linear triangular elements (`nelem, 3, ia[nelem*3]`) and the storage for `id[nnode+1]`, `ik[nnz]` and `sk[nnz]` is allocated. Afterwards, the values of the non-zero elements are calculated and stored in `sk[nnz]`

GetMatrix (nelem, 3, ia, nnode, xc, nnz, id, ik, sk, f)

such that our matrix K is now represented by `id[nnode+1]`, `ik[nnz]` and `sk[nnz]` and the right hand side $\underline{f}$ is calculated using the function **FunctF(x,y)** for describing $\mathbf{f}(x)$. Note, that these two routines are written for general 2d-domains.

The Dirichlet boundary conditions are set in **SetU(nx, ny, u)**. Alternatively, one could use **FunctU(x,y)**. These b.c. are applied in

ApplyDirichletBC(nx, ny, neigh, u, id, ik, sk, f)

via penalty method.

The solver itself is implemented in

JacobiSolve(nnode, id, ik, sk, f, u)

and uses

GetDiag(nnode, id, ik, sk, dd)

to get the diagonal from matrix K. Matrix-times-vector is realized in

CrsMult(w, u, nnode, id, ik, sk) .

A vector $\underline{u}$ can be saved in file named *name* by calling

SaveVector(my_rank,name, u, nx, ny, xl, xr, yb, yt, ierr) .

such that

```
gnuplot jac.dem
```

will give an impression of that vector.

E14 Implement a parallel version of the sequential code !