1 Preliminaries

1.1 MPI on OS-systems LINIX, SOLARIS, AIX, etc.

The **M**essage **P**assing **I**nterface 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 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

1.2 Online help

MPI Homepage	`http://www.mcs.anl.gov/mpi/index.html`
MPI Calls	`http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`
LAM Implementierung	`http://www.lam-mpi.org`
with ftp-download	`http://www.lam-mpi.org/download`

1.3 Getting started on a pool of workstations

- Login at the Pool (computer *myhost*).
- Ensure that you can login from your machine at other ones by

  ```
  rlogin yourhost
  ```

  without being prompted for a passwd.
  If a password is required - enter it, edit/create file *.rhosts*

  ```
  vi ~/.rhosts
  ```

  by adding the line (here, foo names used)

  ```
  myhost  my_user_name
  ```

1.4 Installing the example code

Copy the files *install* and *course.tar.gz* into your working directory, and call `./install`.

1.5 Getting started: LAM-MPI

☐ Define environmental variable *archi* (bash-shell) with one of the systems {LINUX, IRIX, SOLARIS, AIX}.

```
export archi=LINUX
```

or add it in your $(HOME)/.bashrc file. Check with `env|grep archi` .

☐ Initialize the LAM-MPI

```
lamboot
```

This lets you run several processes on your machine (or on a predefined set of machines) in parallel.

☐ If you have your own description file (like *Example/mynode*) of machines able to handle your code, then you could start MPI by

```
lamboot -v mynode
```

This lets you run several processes on your machine (or on a predefined set of machines) in parallel. If `lamboot` reports an error message then try `lamboot -vd mynode` to get a detailed report of the booting in progress.

1.6 Terminating LAM-MPI

☐ Kill all your MPI-processes which did not terminate regularly.

```
lamclean
```

☐ Terminate your MPI session

```
wipe
```

2 Your first parallel code

E1 Compile the program in *Example/firstc* (*Example/firstf*) for archi = LINUX via `make` .
Start the program !

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  -c 4 -lamd  first.LINUX
```

E3 Implement the routines

MPI_Comm_rank and **MPI_Comm_size**,

in order to determine the number of running processes and the local process id. Let the master process (0) write the number of running processes. Start several processes.

E4 Implement the routine

Greetings(myid,numprocs,icomm)

given in *greetings.c* (*greetings.f*). Study the routines

MPI_Send and **MPI_Recv** !

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* does in general not have any information about the length of the data-package to be received.

E6 Write a routine

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

corresponding to E5, 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*.

E7 Test the routines from E5 and E6 first, with two processes. Let process 1 send data and process 0 receive them. Extend the test to several processes.

E8 Combine E5 and E6 to a 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 E5, E6. Test your routines with 2 and more processes !

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)

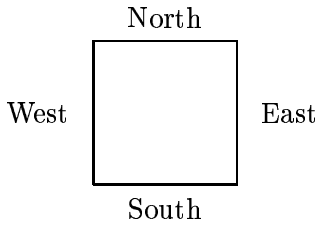
of two Double Precision vectors x and y of local length n. Use

MPI_Allreduce with the operation **MPI_SUM**.

5 Local data exchange

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)

from *example/accuc* (*example/accuf*) 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 !).
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 $nd := (nx+1)*(ny+1)$ local unknowns within one subdomain. We propose a locally rowise ordering of the unknowns. Note, that the global number of unknowns is $N = (procx * nx + 1) * (procy * ny + 1) < procx * procy * nd)$.

GetBound(id,nx,ny,w,s)

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

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

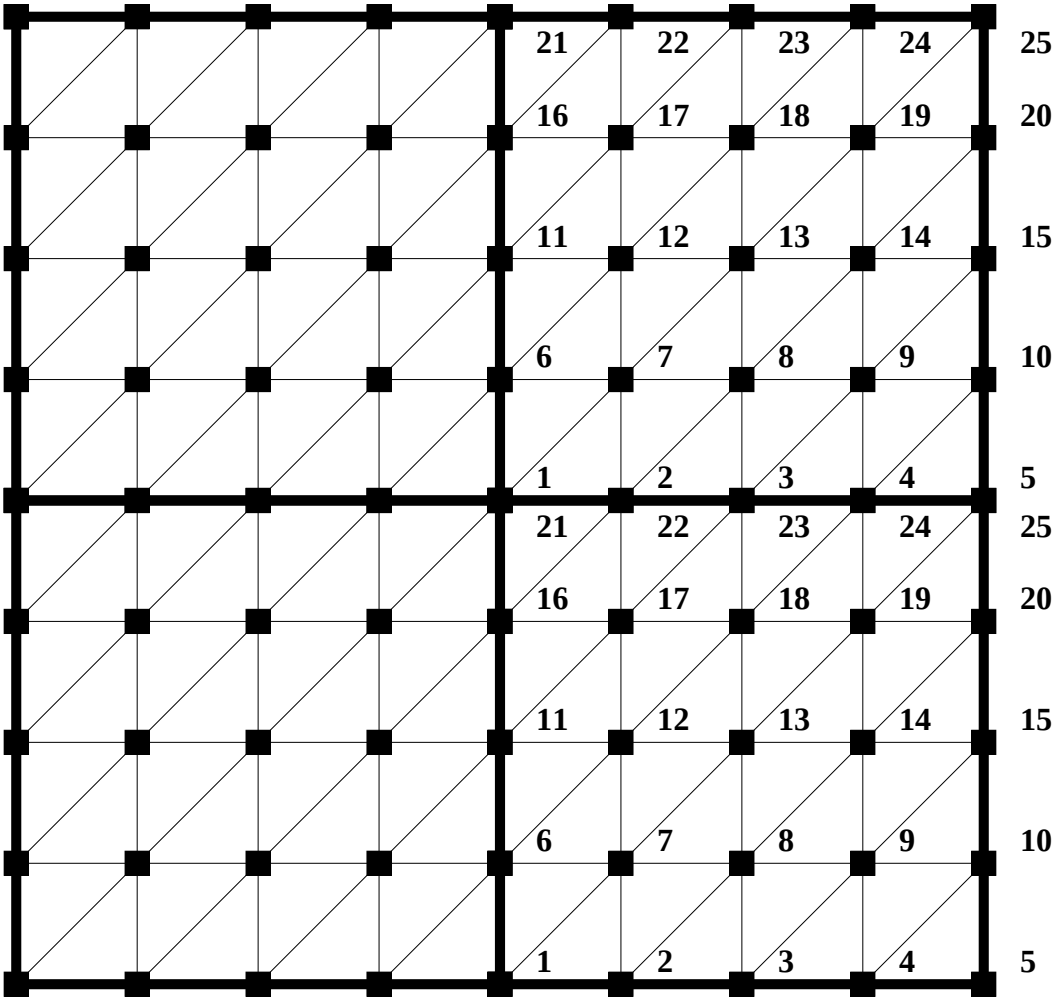


Figure 1: 4 subdomains in local numbering with local discretization $nx = ny = 4$ and global discretization $N_x = N_y = 8$.

adds the values of s to the components of w corresponding to the nodes on the boundary South(id=1), East (id=2), North (id=3), West (id=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,myid,icomm)

where w is both in- and output vector.

6 Iterative 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 *Solution/jacseqc* with the following functions in addition to the functions from P 5.

GetMatrix(nx, ny, xl, xr, yb, yt, sk, id, ik, f)

in which matrix K and right hand side $\underline{f}$ are calculated using the function **FunctF(x,y)** for describing $\mathbf{f}(x)$. Note, that only coordinates and element connectivities are related to the rectangular domain - all other parts in this routine 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, sk, id, ik, f)

via penalty method.
The solver itself is implemented in

JacobiSolve(nx, ny, sk, id, ik, f, u)

and uses

GetDiag(nx, ny, sk, id, d)

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

CrsMult(iza, ize, w, u, id, ik, sk, alfa) .

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

SaveVector(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 !

6.2 Multigrid solver

Here, we will solve (2) by means of a multigrid iteration. You will find a sequential version of the multigrid solver using a ω-Jacobi iterations a smoother and, for simplicity, also as a coarse grid solver in the directory *Solution/mgseqc* . The following functions are added to the functions from previous sections.

AllocLevels(nlevels, nx, ny, pp, ierr)

allocates the memory for all levels and stores the pointers in variable **pp** which is a structure **PointerStruct**. For details see *mg.h* . The matrices and right hand sides will be calculated in

IniLevels(xl, xr, yb, yt, nlevels, neigh, pp, ierr) .

FreeLevels(nlevels, pp, ierr)

frees the allocated memory at the end. Note, that we need for multigrid a different procedure to apply the Dirichlet b.c.

ApplyDirichletBC1(nx, ny, neigh, u, sk, id, ik, f)

The multigrid solver

MGMSolver(pp, crtl, level, neigh)

uses **crtl** as predefined structure **ControlStruct** to control the multigrid cycle (*mg.h*). The same holds for each mg-iteration

MGM(pp, crtl, level, neigh) .

The mg components smoothing, coarse grid solver, interpolation and restriction are defined in

JacobiSmooth(nx, ny, sk, id, ik, f, u, dd, aux, maxiter)

JacobiSolve2(nx, ny, sk, id, ik, f, u, dd, aux)

Interpolate(nx, ny, w, nxc, nyc, wc, neigh)

Restrict(nx, ny, w, nxc, nyc, wc, neigh) .

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

SaveVector(name, pp->u[level], pp->nx[level], pp->ny[level], u, nx, ny, xl, xr, yb, yt, ierr) ,

such that

```
gnuplot mgm.dem
```

will give an impression of that vector.

E15 Implement a parallel version of the sequential code !