Parallel Programming Using MPI

SFU, Jul. 2000

- Introduction:
  - what is MPI?
  - performance of MPI programs:
    is it worth the effort?

- Language Bindings:
  - C and C++
  - F77 and F90

- Monte-Carlo Simulations:
  - $2d$ Ising Model

- Partial Differential Equations:
  - Finite Differences

- Conclusion

Message Passing Interface (MPI)

Central problem in parallel computing:

How can results that were calculated on one processor be accessed by another processor?

$\Rightarrow$ MPI $\Leftarrow$

- standardized message passing system
- set of library routines
- portable
- latest version of the standard: MPI-2; however, available distributions support only MPI-1 with some extensions.

You must change your program.

References

– Using MPI (2nd ed.), W. Gropp, E. Lusk, and A. Skjellum

– MPI – The Complete Reference, Vol. 1 (2nd ed.), M. Snir *et al.*

– http://www.sfu.ca/acs/cluster

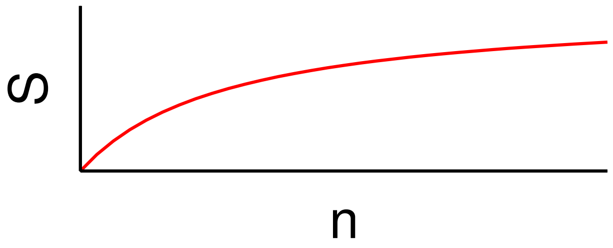
Amdahl's Law

Speedup for parallel execution: $S = T_1/T_n$
(T_n CPU time for n processors).

Estimate for the speedup: $T_n = t_s + t_p/n$
with: t_s cpu time for serial part of the program
t_p cpu time for part executed in parallel

$$S = \frac{T_1}{T_n} = \frac{t_s + t_p}{t_s + t_p/n} \quad \xrightarrow{n\to\infty} \quad \frac{t_s + t_p}{t_s}$$

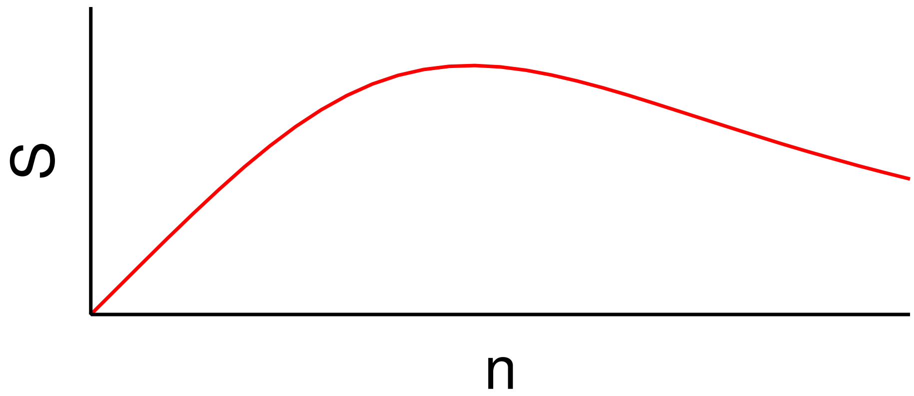
Amdahl's law



more realistic: include cost of communication

$$t_{\mathrm{com}} = t_{cs} + t_{cn}n + t_{cp}n(n-1)$$

$$S = \frac{T_1}{T_n} = \frac{t_s + t_p}{t_s + t_p/n + t_{cs} + t_{cn}n + t_{cp}n(n-1)}$$



Language Bindings

- C
- C++ – defined in MPI-2
- Fortran 77 – yes, but . . .
  - no `include` statement: must copy definition of constants to each routine that uses MPI statements
  - no dynamic memory allocation
    e.g., domain decomposition: size of arrays depends on # of processors that are used, which in turn are specified at run time
    $\longrightarrow$ impossible under F77.
  - advice: use F90
    (you don't need much: `allocate`, `module`, `use`)
- Fortran 90

Generic MPI Program

```
program generic
use mpi
! include 'mpif.h'
integer mpierr,myid,numprocs
...   declarations
call MPI_Init(mpierr)
      first executable statement
call MPI_Comm_rank(MPI_COMM_WORLD,myid,mpierr)
      processor id
call MPI_Comm_size(MPI_COMM_WORLD,numprocs,mpierr)
      # of processors
...
call MPI_Finalize(mpierr)
      last statement
end
```

- all MPI routines are subroutines
- all processors execute the same code
- different results because `myid` is different, e.g.,

```
if (myid == 0) then
    ...
else if (myid == 1) then
    ...
    ...
else
    ...
end if
```

Generic MPI Program

```
...
#include "mpi.h"
int main(int argc, char *argv[]){
...   declarations
int mpierr,myid,numprocs;
mpierr=MPI_Init(&argc,&argv);
      first executable statement
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
      processor id
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
      # of processors
...
MPI_Finalize();
      last statement
}
```

- all MPI routines return int (`mpierr`)
- all processors execute the same code
- different results because `myid` is different, e.g.,

```
if (myid == 0) {
    ...
} else if (myid == 1) {
    ...
    ...
} else {
    ...
}
```

Hello World

```
program hello
include 'mpif.h'
integer mpierr,myid,numprocs
  call MPI_Init(mpierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,myid,mpierr)
  call MPI_Comm_size(MPI_COMM_WORLD,numprocs,mpierr)
  print*,'Hello World, I''m processor ',myid, &
         ' of ',numprocs,'.'
  call MPI_Finalize(mpierr)
end
```

compilation

```
mpif90 hello.f90 -o hello <other compiler options>
```

run on 4 processors

```
mpirun -np 4 ./hello
```

output

```
 Hello World, I'm processor 1 of 4.
 Hello World, I'm processor 0 of 4.
 Hello World, I'm processor 3 of 4.
 Hello World, I'm processor 2 of 4.
```

Hello World

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[]){
int myid,numprocs;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  printf("Hello World, I'm processor %i of %i.\n",
         myid,numprocs);
  MPI_Finalize();
}
```

compilation

```
mpicc hello.c -o hello <other compiler options>
```

run on 4 processors

```
mpirun -np 4 ./hello
```

output

```
Hello World, I'm processor 0 of 4.
Hello World, I'm processor 1 of 4.
Hello World, I'm processor 3 of 4.
Hello World, I'm processor 2 of 4.
```

Monte-Carlo Simulations

Example:

Ising model (describes a magnetic material)

low T: $M = 1$ high T: $M = 0$

$$
\begin{array}{cccccccc}
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow
\end{array}
\qquad\qquad
\begin{array}{cccccccc}
\downarrow & \downarrow & \uparrow & \uparrow & \uparrow & \downarrow & \downarrow & \uparrow \\
\downarrow & \uparrow & \uparrow & \uparrow & \downarrow & \downarrow & \uparrow & \uparrow \\
\uparrow & \uparrow & \uparrow & \downarrow & \downarrow & \uparrow & \downarrow & \uparrow \\
\downarrow & \downarrow & \uparrow & \uparrow & \downarrow & \downarrow & \uparrow & \downarrow
\end{array}
$$

What is the configuration for arbitrary T?

1. start with some initial configuration
2. select particle
3. flip particle according to some probability
   $\rightarrow$ random process
4. iterate 2, 3
5. measure total magnetization M and add to running average: $M_{\text{ave}} = \sum M$.
6. iterate 2 - 5
7. print M_{ave}/(no. of measurements)

```
program ising
include 'mpif.h'
...                                declarations
   call MPI_INIT( mpierr )
   call MPI_COMM_RANK( MPI_COMM_WORLD, myid, mpierr )
   call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, mpierr )
   call pseeds(myid,numprocs)   ! initialize RNG
! master reads input data
   if (myid == 0) then
     read*, l,t,lambda,n_ave
   end if
   call MPI_Bcast(l,1,MPI_INTEGER,0,MPI_COMM_WORLD,mpierr)
   call MPI_Bcast(t,1,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD, &
                  mpierr)
   call MPI_Bcast(lambda,1,MPI_INTEGER,0,MPI_COMM_WORLD,mpierr)
   call MPI_Bcast(n_ave,1,MPI_INTEGER,0,MPI_COMM_WORLD,mpierr)
...                                remaining setup
   n_ave_p=n_ave/numprocs
   do n_upd=1,n_ave_p
     do mcs=1,lambda
...                                Monte-Carlo code
     end do    ! end of one Monte-Carlo step
!  take averages
     rm=dble(2*m)/rll         ! magnetization per spin
     ave_m=ave_m+abs(rm)
...                                other averages
   end do
!  send results to master process
   call MPI_Reduce(ave_m,ave_m_tot,1,MPI_DOUBLE_PRECISION, &
                   MPI_SUM,0,MPI_COMM_WORLD,mpierr)
...                                other  MPI_Sum  reductions
!  write results
   if (myid == 0) then
...                                print results
   end if
   call MPI_FINALIZE(mpierr)
end
```

```
...                                          declarations
#include "mpi.h"
int main(int argc, char *argv[]){
...                                          declarations
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  pseeds(myid,numprocs);
  int l=atoi(argv[1]);
  double t=atof(argv[2]);
  int lambda=atoi(argv[3]);
  int n_ave=atoi(argv[4]);
...                                          remaining setup
  n_ave_p=n_ave/numprocs;
  for (n_upd=1;n_upd<=n_ave_p;n_upd++){
     for (mcs=1;mcs<=lambda;mcs++){
...                                          Monte-Carlo code
     }//end for: mcs
     rm=((double)(2*M))/rll;     // magnetization per spin
     ave_m+=abs(rm);
...                                          other averages
  }//end for: nupd
// send results to master process
  MPI_Reduce(&ave_m,&ave_m_tot,1,MPI_DOUBLE,MPI_SUM,0,
             MPI_COMM_WORLD);
...                                          other MPI_Sum reductions
// write results
  if (myid == 0) {
...                                          print results
  }//end if
  MPI_FINALIZE();
}
```

MPI_Bcast

```
MPI_Bcast(buffer,num,<MPI_Type>,master,comm,mpierr)
<type> buffer(*)
integer num,master,comm,mpierr
```

- broadcasts data from process with id `master` to all other processors
- `<MPI_Type>` is one of `MPI_Integer`, `MPI_Real`, `MPI_Double_precision`, `MPI_Complex`, `MPI_Logical`, `MPI_Character`, etc. (defined in mpif.h)
- `num` successive entries of the type indicated by `<MPI_Type>` are sent starting with the element specified by `buffer`.
- must be called by all processes.

MPI_Reduce

```
MPI_Reduce(sendbuf,recvbuf,num,<MPI_Type>,<op>, &
           master,comm,mpierr)
<type> sendbuf(*),recvbuf(*)
integer num,master,comm,mpierr
```

- all processes send `sendbuf` to `master`, which acts as specified by `op` and stores result in `recvbuf`.
- `<op>` is one of `MPI_Sum`, `MPI_Prod`, `MPI_Max`, `MPI_Min`, etc.
- must be called by all processes.

MPI_Bcast

```
int MPI_Bcast(void* buffer,int num,
              MPI_Datatype <MPI_Type>,int master,
              MPI_Comm comm)
```

- broadcasts data from process with id `master` to all other processors
- `<MPI_Type>` is one of `MPI_Int`, `MPI_Float`, `MPI_Double`, `MPI_Unsigned`, `MPI_Long`, `MPI_Char`, etc. (defined in mpi.h)
- `num` successive entries of the type indicated by `<MPI_Type>` are sent starting with the element specified by `buffer`.
- must be called by all processes.

MPI_Reduce

```
int MPI_Reduce(void* sendbuf,void* recvbuf,int num,
               MPI_Datatype <MPI_Type>,MPI_op <op>,
               int master,MPI_Comm comm)
```

- all processes send `sendbuf` to `master`, which acts as specified by `op` and stores result in `recvbuf`.
- `<op>` is one of `MPI_Sum`, `MPI_Prod`, `MPI_Max`, `MPI_Min`, etc.
- must be called by all processes.

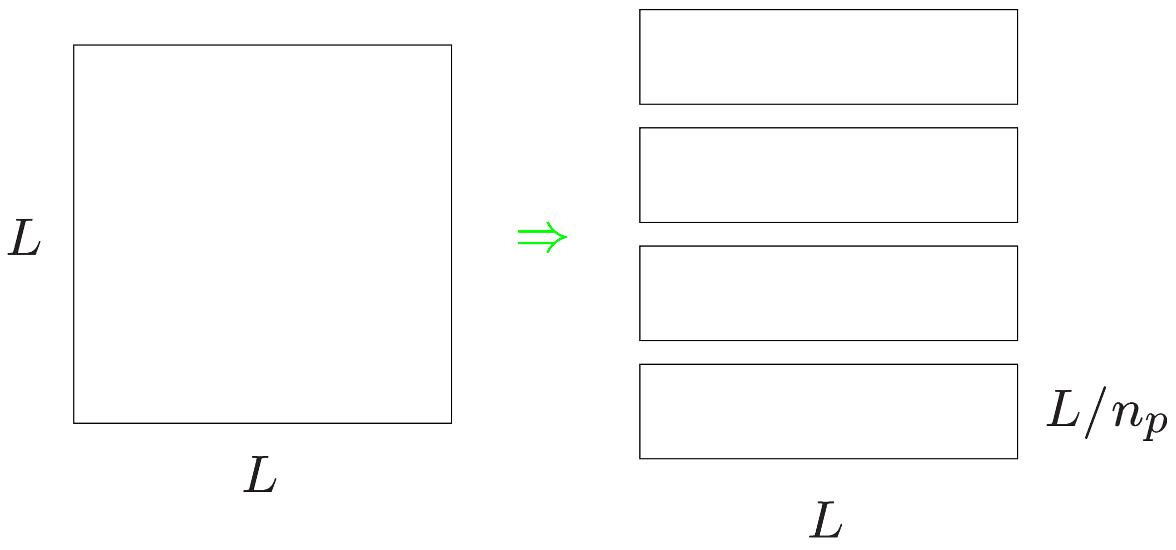
Partial Differential Equations

Domain Decomposition

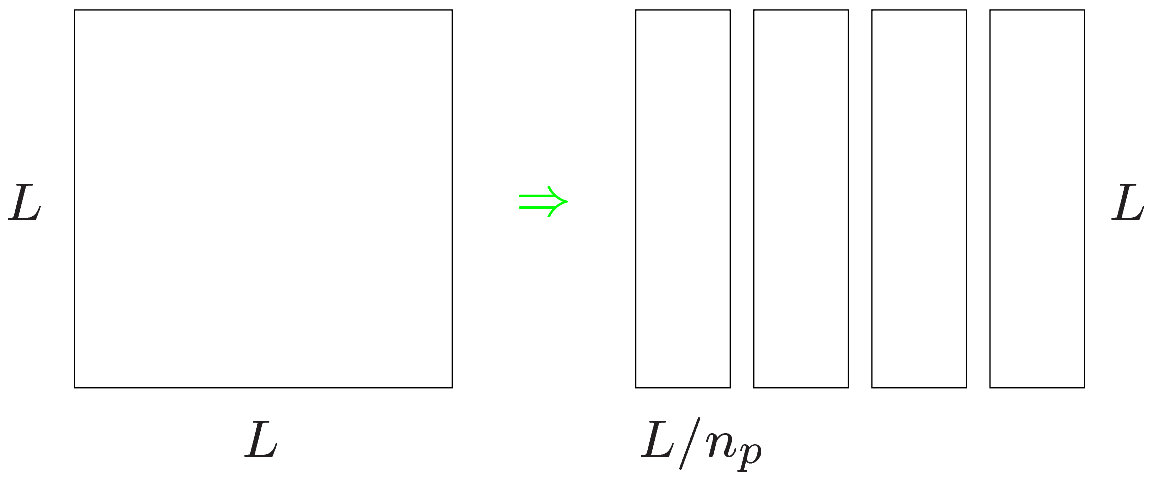
- split system into several parts ($\rightarrow$ domains)
- assign one domain to each processor
- communication required at the boundaries of the domains

System size $L \times L$, n_p processors

C



Fortran



Example: Allan-Cahn equation

$$\frac{\partial f(\mathbf{r},t)}{\partial t} = \nabla^2 f(\mathbf{r},t) + f(\mathbf{r},t) - f(\mathbf{r},t)^3$$

Discretization:

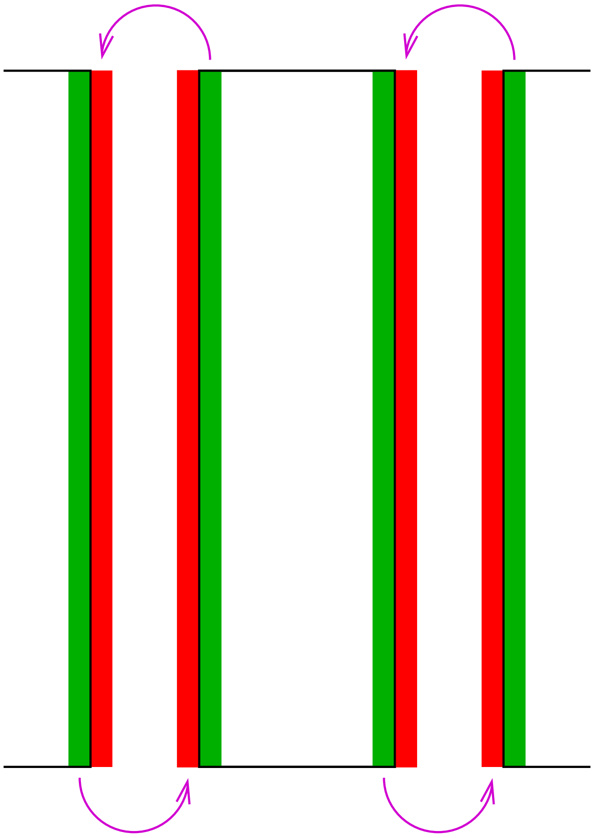
Euler method for time stepping:

$$\frac{\partial f(\mathbf{r},t)}{\partial t} = \text{rhs}(t) \to \frac{f(\mathbf{r},t+\Delta t) - f(\mathbf{r},t)}{\Delta t} = \text{rhs}(t)$$

Finite differences for ∇^2:

$$\nabla^2 f(\mathbf{r},t) \to$$
$$\frac{1}{\Delta x^2}\Big[f(x+\Delta x,y,t) + f(x-\Delta x,y,t)$$
$$+ f(x,y+\Delta x,t) + f(x,y-\Delta x,t) - 4f(x,y,t)\Big]$$

requires communication at the boundaries of the domain:



$\Rightarrow$ two send-receive operations in each timestep.

Example: Allan-Cahn equation

$$\frac{\partial f(\mathbf{r},t)}{\partial t} = \nabla^2 f(\mathbf{r},t) + f(\mathbf{r},t) - f(\mathbf{r},t)^3$$

Discretization:

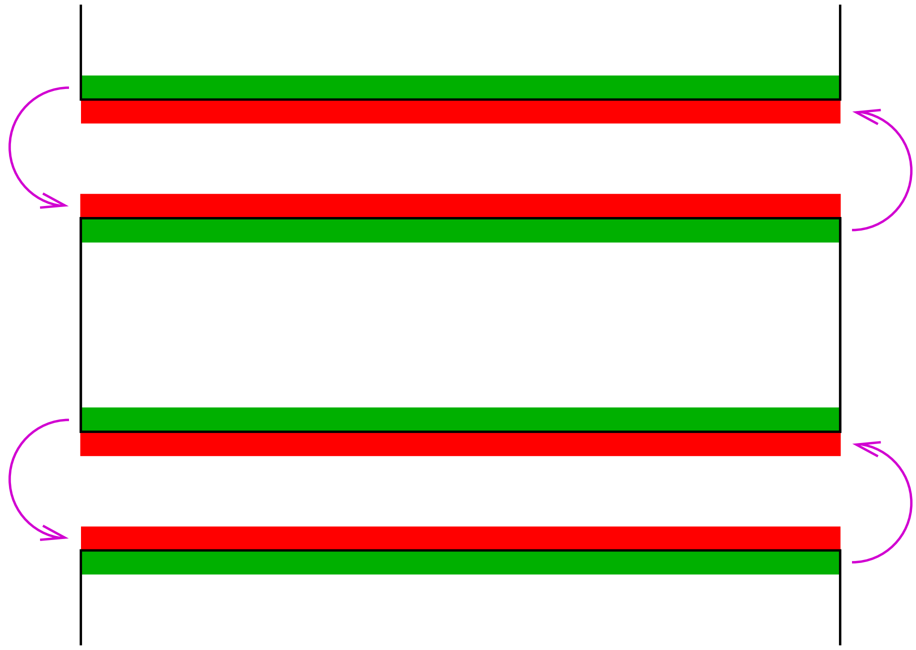
Euler method for time stepping:

$$\frac{\partial f(\mathbf{r},t)}{\partial t} = \mathrm{rhs}(t) \to \frac{f(\mathbf{r},t+\Delta t) - f(\mathbf{r},t)}{\Delta t} = \mathrm{rhs}(t)$$

Finite differences for ∇^2:

$$\nabla^2 f(\mathbf{r},t) \to$$
$$\frac{1}{\Delta x^2}\Big[f(x+\Delta x,y,t) + f(x-\Delta x,y,t)$$
$$+ f(x,y+\Delta x,t) + f(x,y-\Delta x,t) - 4f(x,y,t)\Big]$$

requires communication at the boundaries of the domain:



$\Rightarrow$ two send-receive operations in each time step.

Cost of Communication

No. of data to be transferred in each timestep:

$$N_{\text{trans}} = 2 \times L \times n_p$$

Total CPU-time:

$$t_{\text{cpu}} \sim L^2$$

Fraction transfer-time/cpu-time:

$$t_{\text{com}}/t_{\text{cpu}} \sim n_p/L \quad \xrightarrow{L\to\infty} \quad 0$$

- Domain decomposition becomes more efficient for larger system sizes!

```fortran
module params
  use mpi
  integer lproc,numprocs,myid,jmin,jmax,j1,jl, &
          istat(MPI_STATUS_SIZE)
  ...
end module
...
program allan_cahn
use params
...
   call MPI_Init(mpierr)
   call MPI_Comm_rank(MPI_COMM_WORLD,myid,mpierr)
   call MPI_Comm_size(MPI_COMM_WORLD,numprocs,mpierr)

   if (myid == 0) read*, l,dx,dt,t0,tmax,ntim,···
   call MPI_Bcast(l,1,MPI_INTEGER,0,MPI_COMM_WORLD,mpierr)
   ...
! ensure that l is even and a multiple of numprocs
   if ((l/2)*2 == l) then
     lproc=l/numprocs
   else
     lproc=(l/(2*numprocs))*2
   end if
   l=lproc*numprocs
!  allocate storage
   jmin=myid*lproc
   jmax=(myid+1)*lproc+1
   j1=jmin+1
   jl=jmax-1
   loc_size=lp2*lproc
   allocate(phi(0:lp1,jmin:jmax), ... ,stat=istat_alloc)
   if (istat_alloc > 0) then
     write(0,*) 'error: phi allocation failed'
     call MPI_Abort(MPI_COMM_WORLD,istat_alloc,mpierr)
     stop
   end if
   ...
```

```
...
#include "mpi.h"
int lproc,numprocs,myid,imin,imax,i1,il;
MPI_Status status;
...
int main(int argc,char *argv[]){
...
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   if (myid == 0) scanf("%i%lf%lf%lf%lf%i%i%lf%s%s",
                        &l,&dx,&dt,&t0,&tmax,&ntim, ...);
   MPI_Bcast(&l,1,MPI_INT,0,MPI_COMM_WORLD);
   ...
/* ensure that l is even and a multiple of numprocs */
   if ((l/2)*2 == l) {
     lproc=l/numprocs;
   }else{
     lproc=(l/(2*numprocs))*2;
   }/*end if*/
   l=lproc*numprocs;
/*  allocate storage */
   imin=myid*lproc;
   imax=(myid+1)*lproc+1;
   i1=imin+1;
   il=imax-1;
   loc_size=lp2*lproc;
   phi=alloc_darray2d(imin,imax,0,lp1);
   if (phi == NULL){
     fprintf(stderr," Allocation failure for phi in main\n");
     MPI_Abort(MPI_COMM_WORLD,1);
     exit(1);
   }/*end if*/
...
```

```fortran
  call start(l,phi,stw)        ! initialize phi randomly
  do itim=1,ntim
    call lgv(phi,l,nt(itim))   ! integrate nt(itim) time steps

 ! check for divergence
    phimax=maxval(abs(phi))
    call MPI_Allreduce(phimax,phimax_global,1,       &
                       MPI_DOUBLE_PRECISION,MPI_MAX, &
                       MPI_COMM_WORLD,mpierr)
    div= phimax_global > 10d0
    if (div) exit

 ! calculate Sum(phi(i,j)**2)
    phi2_loc=sum(phi(1:l,j1:jl)*phi(1:l,j1:jl))
    call MPI_Reduce(phi2_loc,phi2_glb,1,MPI_DOUBLE_PRECISION, &
                    MPI_SUM,0,MPI_COMM_WORLD,mpierr)
    if (myid == 0) then
      phi2(itim)=phi2(itim)+phi2_glb
      cpt=etime(cpu)
    end if
    ...
 ! checkpointing: after 2 hours safe data to disk
    call MPI_Bcast(cpt,1,MPI_REAL,0,MPI_COMM_WORLD,mpierr)
    if (cpt-cp0 > 7200. .or. itim == ntim) then
      call save_data(l,t0,tmax,itim, ... , phi)
      cp0=cpt
    end if
  end do

 ! output routines
  call gen_output(s,l,phi2,dt,fout,div)
  if (div) then
    call MPI_Finalize(mpierr)
    stop 'divergence'
  end if
  call MPI_Finalize(mpierr)
end
```

```
  start(l,phi,stw);        /* initialize phi randomly */
  for (itim=1;itim<=ntim;itim++){
    lgv(phi,l,nt[itim]);   /* integrate nt(itim) time steps */

 /* check for divergence */
    phimax=0.;
    for (i=imin;i<=imax;i++) for (j=1;j<=l;j++){
        phimax=max(phimax,fabs(phi[i][j]));
    }/*end for*/
    MPI_Allreduce(&phimax,&phimax_global,1,MPI_DOUBLE,
                  MPI_MAX,MPI_COMM_WORLD);
    div= phimax_global > 10.;
    if (div) break;

 /* calculate Sum(phi(i,j)**2) */
    phi2_loc=0.;
    for (i=i1;i<=il;i++) for (j=1;j<=l;j++){
        phi2_loc+=phi[i][j]*phi[i][j];
    }/*end for*/
    MPI_Reduce(&phi2_loc,&phi2_glb,1,MPI_DOUBLE,MPI_SUM,0,
               MPI_COMM_WORLD);
    if (myid == 0){
      phi2[itim]+=phi2_glb;
      cpt=cpu_time();
    }/*end if*/

 /* checkpointing: after 2 hours safe data to disk */
    MPI_Bcast(&cpt,1,MPI_FLOAT,0,MPI_COMM_WORLD);
    if (cpt-cp0 > 7200. || itim == ntim){
      save_data(l,t0,tmax,itim,stw,nrun,s1,cfout,cfdat,s,phi);
      cp0=cpt;
    }/*end if*/
  }/*end for*/

 /* output routines */
  gen_output(s,l,phi2,dt,cfout,div);
  MPI_Finalize();
}
```

MPI_Send

```
MPI_Send(buf,num,<MPI_Type>,dest,tag,comm,mpierr)
<type> buf(*)
integer num,dest,tag,comm,mpierr
```

- sends data to process with id `dest`
- `num` successive entries of the type indicated by `<MPI_Type>` are sent starting with the element specified by `buf`.
- `tag` is a message identifier.
- this is a **blocking** send, i.e., the program continues only after a matching receive has been posted.

MPI_Send

```
int MPI_Send(void *buf,int num,
             MPI_Datatype <MPI_Type>,int dest,
             int tag,MPI_Comm comm)
```

- sends data to process with id `dest`
- `num` successive entries of the type indicated by `<MPI_Type>` are sent starting with the address specified by `buf`.
- `tag` is a message identifier.
- this is a **blocking** send, i.e., the program continues only after a matching receive has been posted.
- if you want to send a matrix, you must allocate it as a single block of memory.

MPI_Recv

```
MPI_Recv(buf,num,<MPI_Type>,source,tag,comm, &
         status,mpierr)
<type> buf(*)
integer num,source,tag,comm, &
        status(MPI_STATUS_SIZE),mpierr
```

- receives a message sent by the process with id `source` with the message tag `tag`. `num` elements of the type specified by `<MPI_Type>` are stored starting at `buf`.
- `num` must be at least as large as the actual size of the message as specified by the matching `MPI_Send`.
- `status` contains at least 3 elements: `status(MPI_SOURCE)` contains the sender id (useful, if `MPI_ANY_SOURCE` was specified for `source`), `status(MPI_TAG)` contains the message tag (useful, if `MPI_ANY_TAG` was specified for `tag`), and `status(MPI_ERROR)` contains an error code.
- this is a **blocking** receive, i.e., the program will continue only after the messages has been stored in the receive buffer `buf`. The receive can be started whether or not a matching send has been posted.

MPI_Recv

```
int MPI_Recv(void *buf,int num,
             MPI_Datatype <MPI_Type>,int source,
             tag,MPI_Comm comm,MPI_Status *status)
```

- receives a message sent by the process with id `source` with the message tag `tag`. `num` elements of the type specified by `<MPI_Type>` are stored starting at `buf`.
- `num` must be at least as large as the actual size of the message as specified by the matching `MPI_Send`.
- `status` is a structure that contains at least 3 fields: `status.MPI_SOURCE` contains the sender id (useful, if `MPI_ANY_SOURCE` was specified for `source`), `status.MPI_TAG` contains the message tag (useful, if `MPI_ANY_TAG` was specified for `tag`), and `status.MPI_ERROR` contains an error code.
- this is a **blocking** receive, i.e., the program will continue only after the messages has been stored in the receive buffer `buf`. The receive can be started whether or not a matching send has been posted.

Deadlock

data exchange between two processors:

```
if (myid == 0) then
  call MPI_Send(sendbuf,n,MPI_DOUBLE_PRECISION,1,0, &
                MPI_COMMM_WORLD,mpierr)
  call MPI_Recv(recvbuf,n,MPI_DOUBLE_PRECISION,1,1, &
                MPI_COMMM_WORLD,istat,mpierr)
else if (myid == 1) then
  call MPI_Send(sendbuf,n,MPI_DOUBLE_PRECISION,0,1, &
                MPI_COMMM_WORLD,mpierr)
  call MPI_Recv(recvbuf,n,MPI_DOUBLE_PRECISION,0,0, &
                MPI_COMMM_WORLD,istat,mpierr)
end if
```

This may hang forever!

Circular data exchange:

```
id_s=modulo(myid+1,numprocs)
id_r=modulo(myid-1,numprocs)
if (myid == 0) then
  call MPI_Recv(recvbuf,n,MPI_DOUBLE_PRECISION,id_r,id_r, &
                MPI_COMMM_WORLD,istat,mpierr)
  call MPI_Send(sendbuf,n,MPI_DOUBLE_PRECISION,id_s,myid, &
                MPI_COMMM_WORLD,mpierr)
else
  call MPI_Send(sendbuf,n,MPI_DOUBLE_PRECISION,id_s,myid, &
                MPI_COMMM_WORLD,mpierr)
  call MPI_Recv(recvbuf,n,MPI_DOUBLE_PRECISION,id_r,id_r, &
                MPI_COMMM_WORLD,istat,mpierr)
end if
```

Safe!

Deadlock

data exchange between two processors:

```
if (myid == 0) {
  MPI_Send(sendbuf,n,MPI_DOUBLE,1,0,MPI_COMMM_WORLD);
  MPI_Recv(recvbuf,n,MPI_DOUBLE,1,1,MPI_COMMM_WORLD,istat);
} else if (myid == 1) {
  MPI_Send(sendbuf,n,MPI_DOUBLE,0,1,MPI_COMMM_WORLD);
  MPI_Recv(recvbuf,n,MPI_DOUBLE,0,0,MPI_COMMM_WORLD,istat);
}
```

This may hang forever!

Circular data exchange:

```
id_s=(myid+1) % numprocs;
id_r=(myid-1+numprocs) % numprocs;
if (myid == 0) {
  MPI_Recv(recvbuf,n,MPI_DOUBLE,id_r,id_r,MPI_COMMM_WORLD,
           istat);
  MPI_Send(sendbuf,n,MPI_DOUBLE,id_s,myid,MPI_COMMM_WORLD);
} else {
  MPI_Send(sendbuf,n,MPI_DOUBLE,id_s,myid,MPI_COMMM_WORLD);
  MPI_Recv(recvbuf,n,MPI_DOUBLE,id_r,id_r,MPI_COMMM_WORLD,
           istat);
}
```

Safe!

MPI_Sendrecv

```
MPI_Sendrecv(sendbuf,num_s,<MPI_Type_s>, &
             dest,sendtag,               &
             recvbuf,num_r,<MPI_Type_r>, &
             source,recvtag,             &
             comm,status,mpierr)
<type_s> sendbuf(*)
<type_r> recvbuf(*)
integer num_s,source,sendtag,num_r,dest,recvtag, &
        comm,status(MPI_STATUS_SIZE),mpierr
```

- combination of `MPI_Send` and `MPI_Recv`
- send and receive happen “simultanously”.
- cycles in communication pattern do not lead to deadlock.

MPI_Sendrecv

```
int MPI_Sendrecv(void *sendbuf,int num_s,
                 MPI_Datatype <MPI_Type_s>,
                 int dest,int sendtag,
                 void *recvbuf,int num_r,
                 MPI_Datatype <MPI_Type_r>,
                 int source,int recvtag,
                 MPI_Comm comm,MPI_Status *status)
```

- combination of `MPI_Send` and `MPI_Recv`
- send and receive happen "simultanously".
- cycles in communication pattern do not lead to deadlock.

```
subroutine start(l,phi,stw)
! randomly initialize the order parameter field phi
use params
double precision phi(0:lp1,jmin:jmax)
...
   call randv(phi,l,stw)      ! fill phi with random numbers
! get the boundary rows:
! first: send left column to next higher processor,
!        receive right column from next lower processor
   isendto=modulo(myid+1,numprocs)
   irecvfrom=modulo(myid-1,numprocs)
   msendtag=myid*numprocs+isendto
   mrecvtag=irecvfrom*numprocs+myid
   call MPI_Sendrecv(phi(1,jl),l,MPI_DOUBLE_PRECISION,     &
                     isendto,msendtag,                     &
                     phi(1,jmin),l,MPI_DOUBLE_PRECISION,   &
                     irecvfrom,mrecvtag,                   &
                     MPI_COMM_WORLD,istat,mpierr)
! second: send right column to next lower processor,
!         receive left column from next higher processor
   idmy=isendto
   isendto=irecvfrom
   irecvfrom=idmy
   msendtag=myid*numprocs+isendto
   mrecvtag=irecvfrom*numprocs+myid
   call MPI_Sendrecv(phi(1,j1),l,MPI_DOUBLE_PRECISION,     &
                     isendto,msendtag,                     &
                     phi(1,jmax),l,MPI_DOUBLE_PRECISION,   &
                     irecvfrom,mrecvtag,                   &
                     MPI_COMM_WORLD,istat,mpierr)

! correct the periodic boundary conditions
   do j=jmin,jmax
     phi(0,j)=phi(l,j)
     phi(lp1,j)=phi(1,j)
   end do
   return
end
```

```
void start(int l, double **phi, double stw){
/* randomly initialize the order parameter field phi */
...
   randv(phi,l,stw);
/* get the boundary rows:
   first: send upper row to next higher processor,
          receive lower row from next lower processor  */
   isendto=(myid+1) % numprocs;
   irecvfrom=(myid-1+numprocs) % numprocs;
   msendtag=myid*numprocs+isendto;
   mrecvtag=irecvfrom*numprocs+myid;
   MPI_Sendrecv(&phi[i1][1],l,MPI_DOUBLE,isendto,msendtag,
                &phi[imin][1],l,MPI_DOUBLE,irecvfrom,mrecvtag,
                MPI_COMM_WORLD,&status);
/* second: send lower row to next lower processor,
           receive upper row from next higher processor */
   idmy=isendto;
   isendto=irecvfrom;
   irecvfrom=idmy;
   msendtag=myid*numprocs+isendto;
   mrecvtag=irecvfrom*numprocs+myid;
   MPI_Sendrecv(&phi[i1][1],l,MPI_DOUBLE,isendto,msendtag,
                &phi[imax][1],l,MPI_DOUBLE,irecvfrom,mrecvtag,
                MPI_COMM_WORLD,&status);

/* correct the periodic boundary conditions */
   for (i=imin;i<=imax;i++){
     phi[i][0]=phi[i][l];
     phi[i][lp1]=phi[i][1];
   }
}
```

```
subroutine lgv(phi,l,nt)
!
!  integrate Allan-Cahn equation from t to t+nt*dt using
!  the Euler method.
!
use params
implicit none
double precision phi(0:lp1,jmin:jmax)
integer l,nt
double precision, allocatable :: tmp(:,:)
integer it,i,j,idummy
integer isendto,irecvfrom,msendtag,mrecvtag,mpierr
double precision d2phi

 allocate(tmp(l,j1:jl))

 do it=1,nt

!  The next do-loop calculates the magnetization at time t+dt.

   do j=j1,jl
     do i=1,l

!  calculation of the laplacian of phi(x,y):
       d2phi=phi(i+1,j)+phi(i-1,j)+phi(i,j+1)+phi(i,j-1) &
             -4d0*phi(i,j)

!  magnetization at t+dt
       tmp(i,j)=phi(i,j)+dtdx2*d2phi                      &
                +dt*phi(i,j)*(1d0-phi(i,j)*phi(i,j))
     end do
   end do
   phi(1:l,j1:jl)=tmp
```

```
void lgv(double **phi,int l,int nt){
/*
   integrate the Allan-Cahn equation from t to t+nt*dt using
   the Euler method.
*/
double **tmp;
int it,i,j,idummy;
int isendto,irecvfrom,msendtag,mrecvtag;
double d2phi;
 tmp=alloc_darray2d(i1,il,1,l);
 for (it=1;it<=nt;it++){
/*
   The next loop calculates the magnetization at time t+dt.
*/
   for (i=i1;i<=il;i++){
     for (j=1;j<=l;j++){
/*
   calculation of the laplacian of phi(x,y):
*/
       d2phi=phi[i+1][j]+phi[i-1][j]+phi[i][j+1]+phi[i][j-1]
             -4.*phi[i][j];
       tmp[i][j]=phi[i][j]+dtdx2*d2phi
                 +dt*phi[i][j]*(1.-phi[i][j]*phi[i][j]);
     }/*end for*/
   }/*end for*/
   for (i=i1;i<=il;i++){
     for (j=1;j<=l;j++){
       phi[i][j]=tmp[i][j];
     }/*end for*/
   }/*end for*/
```

```
! get the boundary rows:
! first: send left column to next higher processor,
!        receive right column from next lower processor

   isendto=modulo(myid+1,numprocs)
   irecvfrom=modulo(myid-1,numprocs)
   msendtag=myid*numprocs+isendto
   mrecvtag=irecvfrom*numprocs+myid
   call MPI_Sendrecv(phi(1,jl),l,MPI_DOUBLE_PRECISION,    &
                     isendto,msendtag,                    &
                     phi(1,jmin),l,MPI_DOUBLE_PRECISION,  &
                     irecvfrom,mrecvtag,                  &
                     MPI_COMM_WORLD,istat,mpierr)

! second: send right column to next lower processor,
!         receive left column from next higher processor

   idummy=isendto
   isendto=irecvfrom
   irecvfrom=idummy
   idmy=msendtag
   msendtag=myid*numprocs+isendto
   mrecvtag=irecvfrom*numprocs+myid
   call MPI_Sendrecv(phi(1,j1),l,MPI_DOUBLE_PRECISION,     &
                     isendto,msendtag,                     &
                     phi(1,jmax),l,MPI_DOUBLE_PRECISION,   &
                     irecvfrom,mrecvtag,                   &
                     MPI_COMM_WORLD,istat,mpierr)

! correct the periodic boundary conditions
   do j=jmin,jmax
     phi(0,j)=phi(l,j)
     phi(lp1,j)=phi(1,j)
   end do
 end do
 deallocate(tmp)
return
end
```

```
/*  get the boundary rows:
    first: send left column to next higher processor,
           receive right column from next lower processor
*/
   isendto=(myid+1) % numprocs;
   irecvfrom=(myid-1+numprocs) % numprocs;
   msendtag=myid*numprocs+isendto;
   mrecvtag=irecvfrom*numprocs+myid;
   MPI_Sendrecv(&phi[il][1],l,MPI_DOUBLE,isendto,msendtag,
                &phi[imin][1],l,MPI_DOUBLE,irecvfrom,mrecvtag,
                MPI_COMM_WORLD,&status);

/*  second: send right column to next lower processor,
            receive left column from next higher processor
*/
   idmy=isendto;
   isendto=irecvfrom;
   irecvfrom=idmy;
   idmy=msendtag;
   msendtag=myid*numprocs+isendto;
   mrecvtag=irecvfrom*numprocs+myid;
   MPI_Sendrecv(&phi[i1][1],l,MPI_DOUBLE,isendto,msendtag,
                &phi[imax][1],l,MPI_DOUBLE,irecvfrom,mrecvtag,
                MPI_COMM_WORLD,&status);
/* correct the periodic boundary conditions */
   for (i=imin;i<=imax;i++){
     phi[i][0]=phi[i][l];
     phi[i][lp1]=phi[i][1];
   }/*end for*/
 }/*end for*/
 free_darray2d(tmp,i1,il,1,l);
}
```

Summary

- not much is needed: 6 functions
  `MPI_Init`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Send`, `MPI_Recv`, `MPI_Finalize`

- much more is available: `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Sendrecv`, `MPI_Abort`

- and much more:
  - nonblocking communication: `MPI_Isend`, `MPI_Irecv`, `MPI_Wait`, ...
  - different topologies: `MPI_Cart_create`, etc.
  - global communication: `MPI_Gather`, `MPI_Scatter`, `MPI_Allgather`, `MPI_Alltoall`, etc.
  - communicators: `MPI_Comm_dup`, `MPI_Comm_create`, `MPI_Comm_split`, ...

...