Introduction to MPI

Markus Uhlmann

CIEMAT

www.ciemat.es/sweb/comfos/personal/uhlmann

Universidad Carlos III de Madrid

May 2007

Outline

1. Introduction
2. MPI Program Structure — `hello world`
3. Point-to-point Communication — `deadlock` `non_blocking`
4. Collective Communication — π
5. Case Study: 2D Poisson Problem — `jacobi`
6. Extensions

1. Message Passing Model

- each process works on LOCAL data
  & DIFFERENT instructions

- EXPLICIT interchange of data by function calls

- exchange is MUTUAL process

$\Rightarrow$ flexibility, performance

BUT: time-consuming implementation

What is MPI?

Message Passing Interface

- library of communication subroutines
- implementation defined in standards MPI-1, MPI-2
- well adapted to machines with distributed memory
- wide availability (also: shared memory and hybrid)
- interfaces for `C` or `FORTRAN`

Comparison of Paradigms

Message Passing (MPI)

- efficiency
- portability
- flexibility

Multithreading (OpenMP)

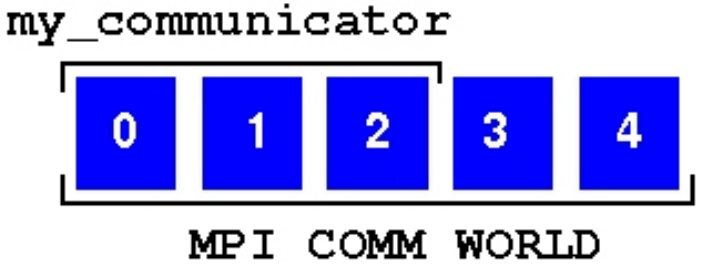
- simplicity

MPI Communicators

- 'handles' representing a group of processes
- communicate only among members of same communicator
- each process has a unique rank within each communicator

$$0 \leq \texttt{rank} \leq \texttt{no. of processes - 1}$$

- a process can belong to different communicators
- default global communicator: `MPI_COMM_WORLD`



MPI Basic Syntax

- naming convention: routines/functions, constants

  `MPI_XXXXX`

- return values: integer

```
ierr=MPI_Finalize();
```

```
CALL MPI_FINALIZE(ierr)
```

- data types: defined in transparent way for the user

`MPI_LONG`	`MPI_INTEGER`
`MPI_FLOAT`	`MPI_REAL`
`MPI_DOUBLE`	`MPI_DOUBLE_PRECISION`
`MPI_CHAR`	`MPI_CHARACTER ...`

2. MPI Program Structure

1. include MPI header file
2. variable declarations
3. initialize the MPI environment
4. ...do computation and MPI communication calls...
5. close MPI communications

Example "Hello World" (FORTRAN)

```
      PROGRAM HELLO
      INCLUDE 'mpif.h'
      INTEGER myrank, size, ierr
C     Initialize MPI:
      call MPI_INIT(ierr)
C     Get my rank:
      call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
C     Get the total number of processors:
      call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
      WRITE(*,*)"Processor",myrank,"of ",size, ": Hello World!"
C     Terminate MPI
      call MPI_FINALIZE(ierr)
      END
```

(source)

Example "Hello World" (C)

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {
    int myrank, size;
/* Initialize MPI       */
    MPI_Init(&argc, &argv);
/* Get my rank          */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
/* Get the total number of processors */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Processor %d of %d: Hello World!\n", myrank, size);
/* Terminate MPI        */
    MPI_Finalize();
   }
```

(source)

Compiling & Executing MPI Programs

Compilation

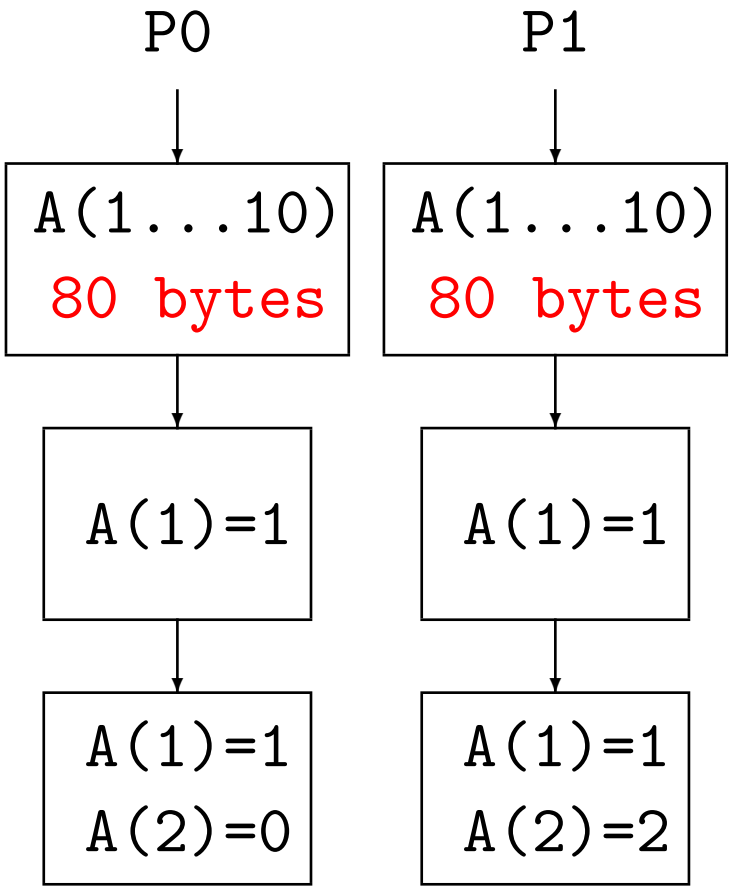
- supply the name of the library explicitly
  `f77 <objects> -lmpi`
- use script with pre-defined variables
  `mpif77`, `mpif90`, `mpicc`

Execution

- use start-up script of MPI implementation, e.g.
  `mpirun -np <# procs> <executable>`
- run on single processor: `-all-local`

Memory Allocation & Access Is LOCAL

```
c  allocates 10 x real, locally
c  -> global_size=numprocs*10*64bit
   real*8 A(10)
     ...
c  process-independent assignment
   A(1)=1.0
     ...
c  assignment with local value
   call MPI_COMM_RANK(MPI_COMM_WORLD,
&            myrank,ierr)
   A(2)=dfloat(myrank)*2.0
```



3. Point-to-point Communication

basic communication between PAIRS of processors

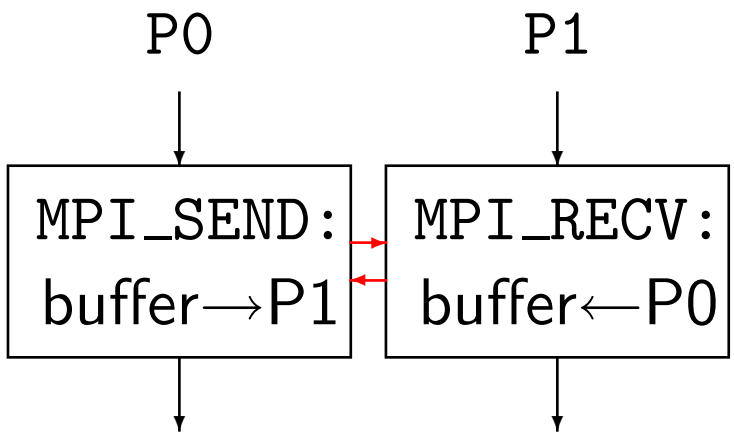
$$\boxed{\text{message} = \text{envelope} + \text{body}}$$

- source process id
- destination process id
- communicator id
- tag (to classify messages)

- buffer - the message data
- datatype of message data
- count (number of items)

Example: Simple Send & Receive

```
 ...
 integer istat(MPI_STATUS_SIZE)
 ...
 isource=0
 idest=1
 itag=99
 icount=10
 if(myrank.eq.isource)then
    call MPI_SEND(buffer,icount,MPI_REAL,
&    idest,itag,MPI_COMM_WORLD,ierr)
 elseif (myrank.eq.idest)then
    call MPI_RECV(buffer,icount,MPI_REAL,
&    isource,itag,MPI_COMM_WORLD,istat,ierr)
 endif
 ...
```



(source)

What happens when Sending/Receiving?

- "blocking" operations until completion
- `MPI_RECV` completion:
  $\longrightarrow$ a message with matching envelope was received
  $\Rightarrow$ data in buffer is available
- `MPI_SEND` completion:
  $\longrightarrow$ message is handed off to MPI
  (either copied to internal buffer or already transferred)
  $\Rightarrow$ buffer can be overwritten

$\rightsquigarrow$ careful organisation of communication patterns!

Send-Receive: Matching of Messages

```
myrank1:                                  myrank2:
MPI_SEND(buffer,icount1,                  MPI_RECV(buffer,icount2,
         dtype1,idest,itag1,                       dtype2,isource,itag2,
               icomm1,ierr)                              icomm2,istat,ierr)
```

- isource = myrank1 OR isource = MPI_ANY_SOURCE
- idest = myrank2
- itag2 = itag1 OR itag2 = MPI_ANY_TAG
- icomm2 = icomm1 AND icount2 $\geq$ icount1
- datatypes are not checked!

Status of Completed Receive Operation

```
integer istat(MPI_STATUS_SIZE)
MPI_RECV(buffer,icount,dtype,MPI_ANY_SOURCE,MPI_ANY_TAG,
                                              icomm,istat,ierr)
```

- entries of status array after completion:

  `istat(MPI_SOURCE)` $\longrightarrow$ source process id

  `istat(MPI_TAG)` $\longrightarrow$ message 'tag'

  `istat(MPI_ERROR)` $\longrightarrow$ error code

  `call MPI_GET_COUNT(istat,dtype,icount2)`

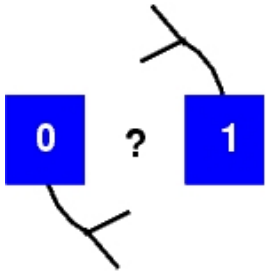
  $\longrightarrow$ `icount2` is the number of received entries

Safe Communication


```
 if      (myrank.eq.0)then
      call MPI_SEND(buffer1,icount,MPI_REAL,1,itag1,
&          MPI_COMM_WORLD,ierr)
      call MPI_RECV(buffer2,icount,MPI_REAL,1,itag2,
&          MPI_COMM_WORLD,istat,ierr)
 elseif (myrank.eq.1)then
      call MPI_RECV(buffer2,icount,MPI_REAL,0,itag1,
&          MPI_COMM_WORLD,istat,ierr)
      call MPI_SEND(buffer1,icount,MPI_REAL,0,itag2,
&          MPI_COMM_WORLD,ierr)
 endif
```

(source)

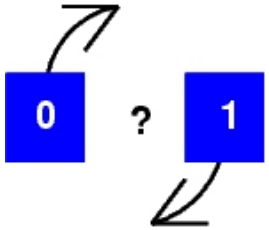
Communication Deadlock




```
 if      (myrank.eq.0)then
      call MPI_RECV(buffer2,icount,MPI_REAL,1,itag,
&          MPI_COMM_WORLD,istat,ierr)
      call MPI_SEND(buffer1,icount,MPI_REAL,1,itag,
&          MPI_COMM_WORLD,ierr)
 elseif (myrank.eq.1)then
      call MPI_RECV(buffer2,icount,MPI_REAL,0,itag,
&          MPI_COMM_WORLD,istat,ierr)
      call MPI_SEND(buffer1,icount,MPI_REAL,0,itag,
&          MPI_COMM_WORLD,ierr)
 endif
```

(source)

Possible Communication Deadlock




```
 if      (myrank.eq.0)then
      call MPI_SEND(buffer1,icount,MPI_REAL,1,itag,
&          MPI_COMM_WORLD,ierr)
      call MPI_RECV(buffer2,icount,MPI_REAL,1,itag,
&          MPI_COMM_WORLD,istat,ierr)
 elseif (myrank.eq.1)then
      call MPI_SEND(buffer1,icount,MPI_REAL,0,itag,
&          MPI_COMM_WORLD,ierr)
      call MPI_RECV(buffer2,icount,MPI_REAL,0,itag,
&          MPI_COMM_WORLD,istat,ierr)
 endif
```

(source)

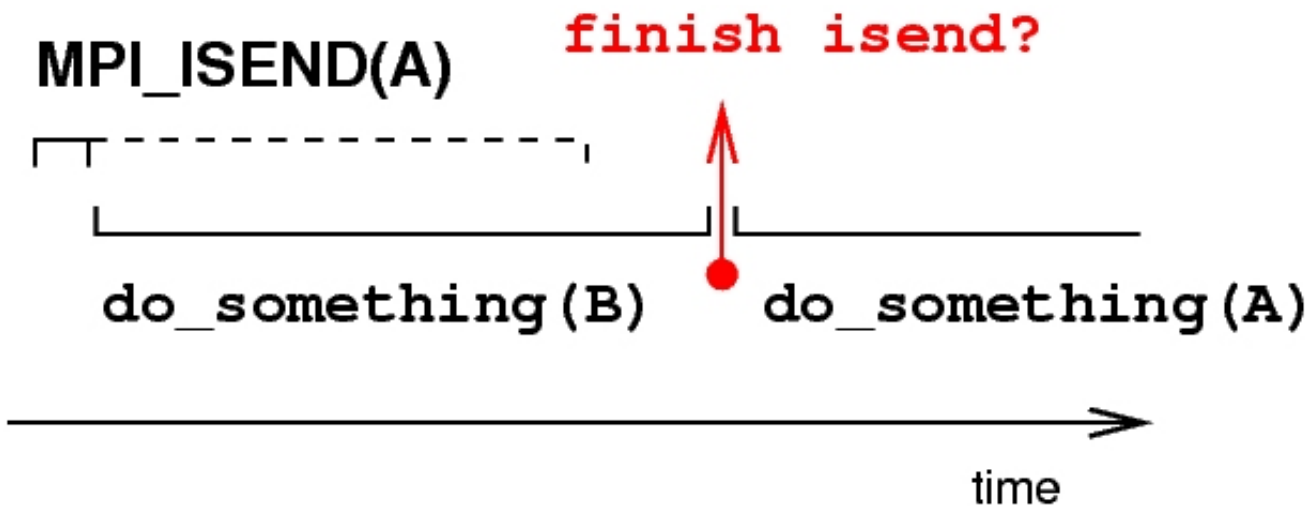
Send Modes

- standard mode `MPI_SEND`
  $\longrightarrow$ direct transmission or buffering (depends on size)
- synchronous mode `MPI_SSEND`
  $\longrightarrow$ completes when reception has begun
- buffered mode `MPI_BSEND`
  $\longrightarrow$ always buffering
- ready mode `MPI_RSEND`
  $\longrightarrow$ assumes matching `MPI_RECV` has been posted

(source)

Non-blocking Communication

- overlapping communication & computation




```
call MPI_ISEND(A,...,irequest)
call do_something(B)
call MPI_WAIT(irequest,...)
call do_something(A)
```

Non-blocking Communication Syntax

- send/receive operations:

```
MPI_ISEND(buffer,count,dtype,dest,tag,comm, request,err)
MPI_IRECV(buffer,count,dtype,source,tag,comm, request,err)
```

`request` → 'handle' identifying the posted operation

- waiting for completion:

```
MPI_WAIT(request,status,err)
```

- testing for completion:

```
MPI_TEST(request,flag,status,err)
```

`flag` → 'logical' indicating completion

Non-blocking Communication Example

```
 if      (myrank.eq.0)then
      call MPI_IRECV(buffer2,icount,MPI_REAL,1,itag,
&          MPI_COMM_WORLD,irequest,ierr)
      buffer1(1)=0.0
      call MPI_WAIT(irequest,istatus,ierr)
      buffer2(icount)=buffer2(icount)*0.25
 elseif (myrank.eq.1)then
      call MPI_ISEND(buffer1,icount,MPI_REAL,0,itag,
&          MPI_COMM_WORLD,irequest,ierr)
      buffer2(4)=1.5
      call MPI_WAIT(irequest,istatus,ierr)
      buffer1(icount)=0.0
 endif
```

(source)

The 10 Essential MPI Statements

(1) `include 'mpif.h'`

(2) `integer istat(MPI_STATUS_SIZE)`

(3) `call MPI_INIT(...)`

(4) `call MPI_COMM_RANK(...)`

(5) `call MPI_COMM_SIZE(...)`

(6) `call MPI_[I]SEND(...)`

(7) `call MPI_[I]RECV(...)`

(8) `call MPI_WAIT(...)`

(9) `call MPI_BARRIER(...)`

(10) `call MPI_FINALIZE(...)`

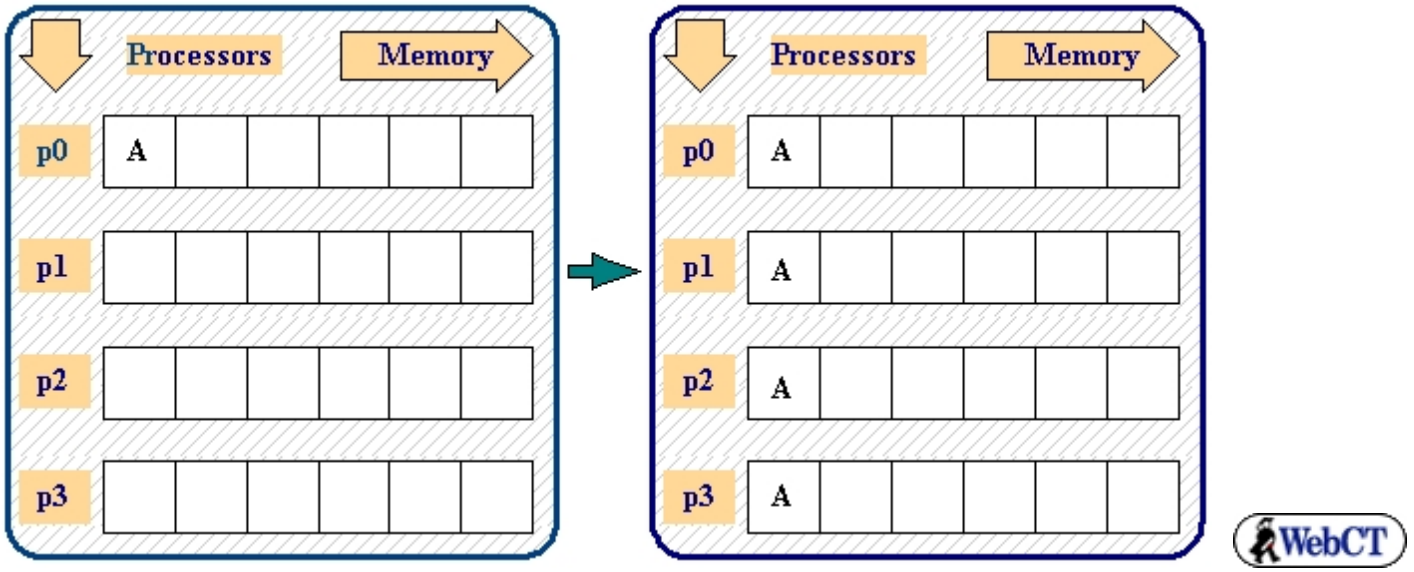
4. Collective Communication

routines which are defined for convenience

- broadcasting of data
- gathering of data
- scattering of data
- reduction operations
- barrier synchronization

$\Rightarrow$ all processors in a communicator need to participate!

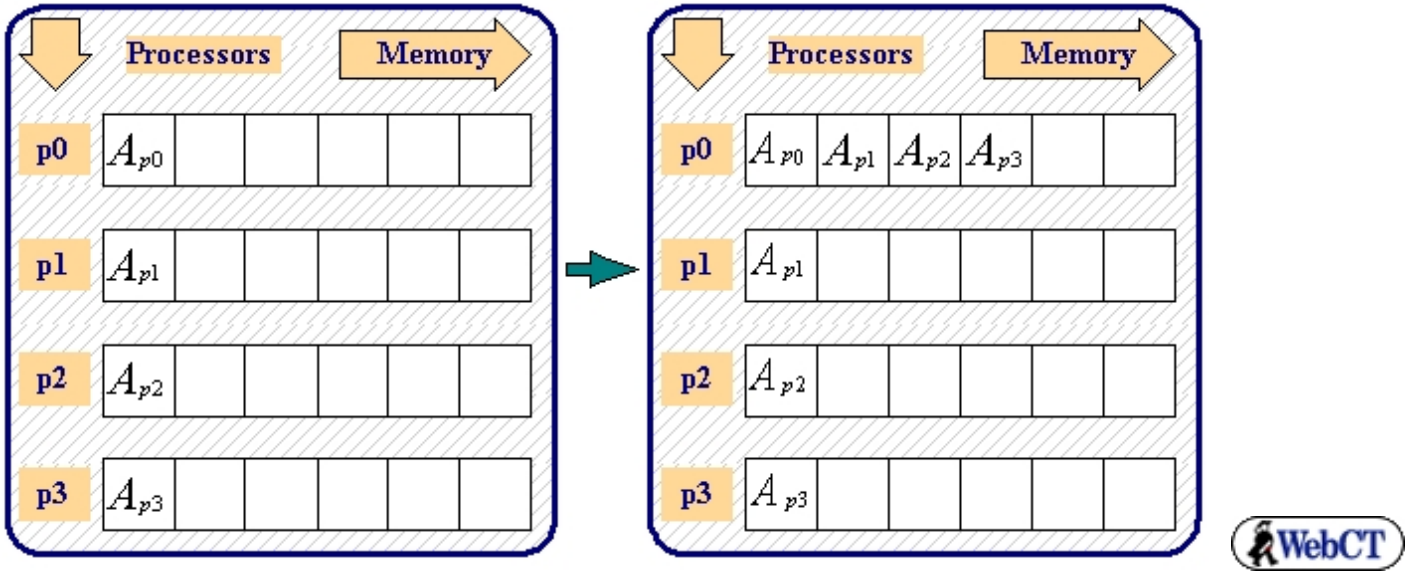
Broadcast Operation




```
MPI_BCAST(buffer,icount,dtype,iroot,icomm,ierr)
```

- one-to-all communication operation (source)

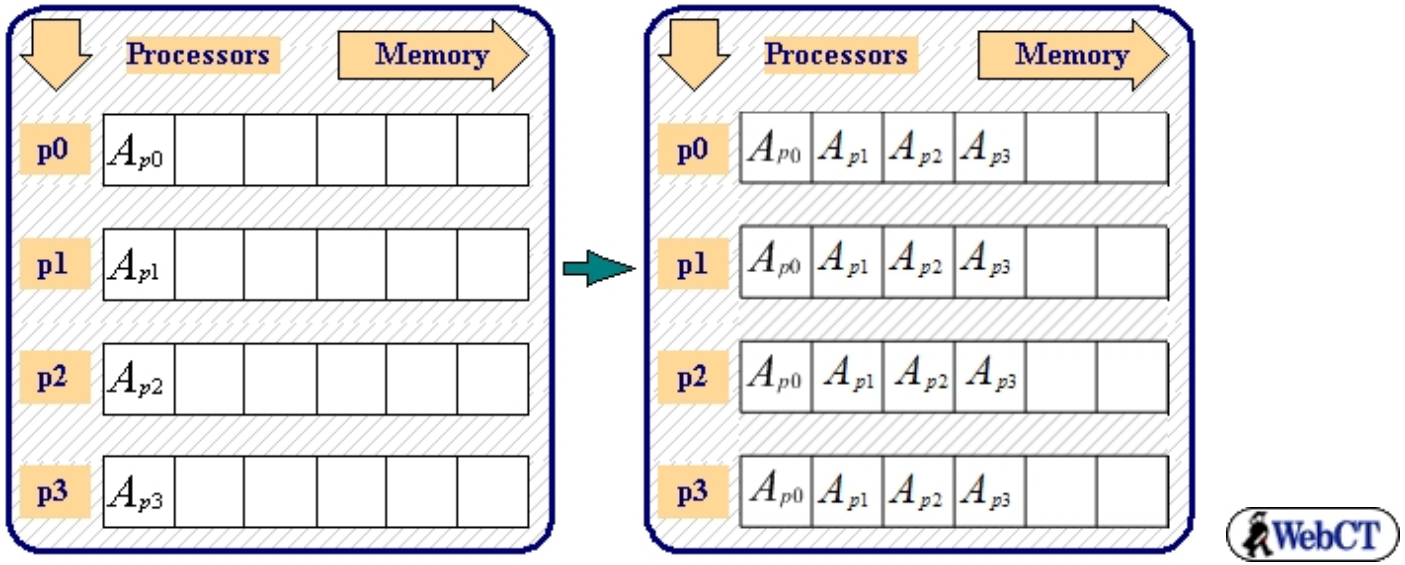
Gather Operation




```
MPI_GATHER(sendbuff,sendcount,sendtype,
           recvbuff,recvcount,recvtype,iroot,icomm,ierr)
```

- all-to-one communication operation (source)

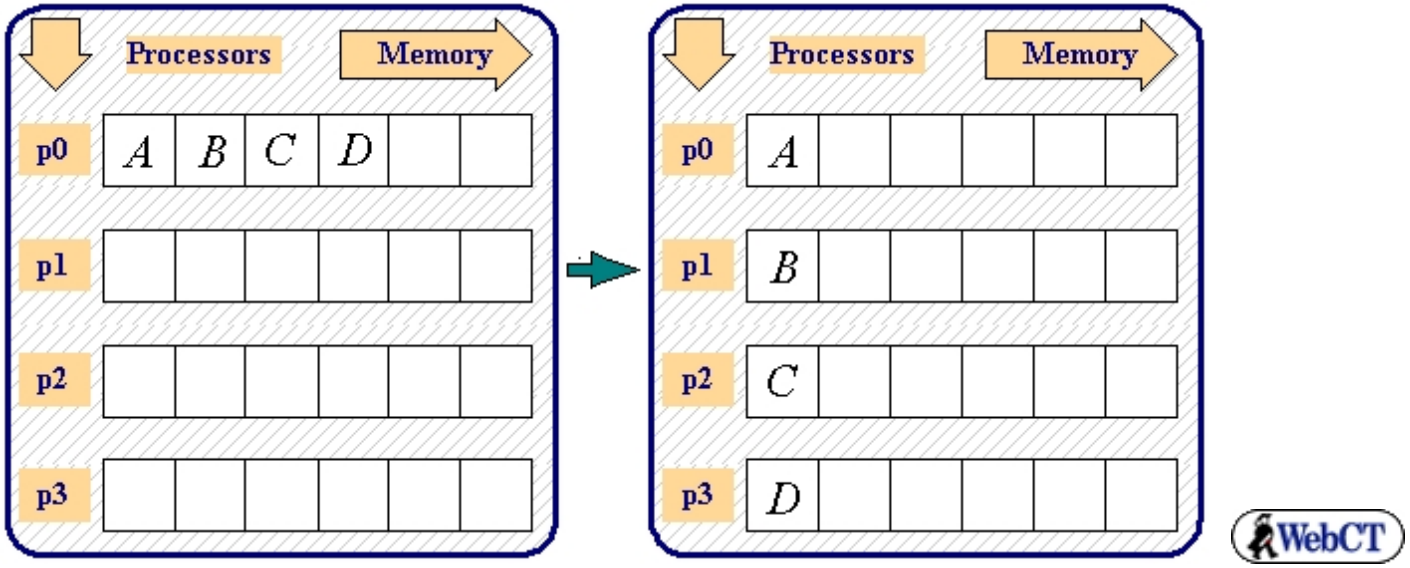
Allgather Operation




```
MPI_ALLGATHER(sendbuff,sendcount,sendtype,
              recvbuff,recvcount,recvtype,icomm,ierr)
```

- all-to-all communication operation

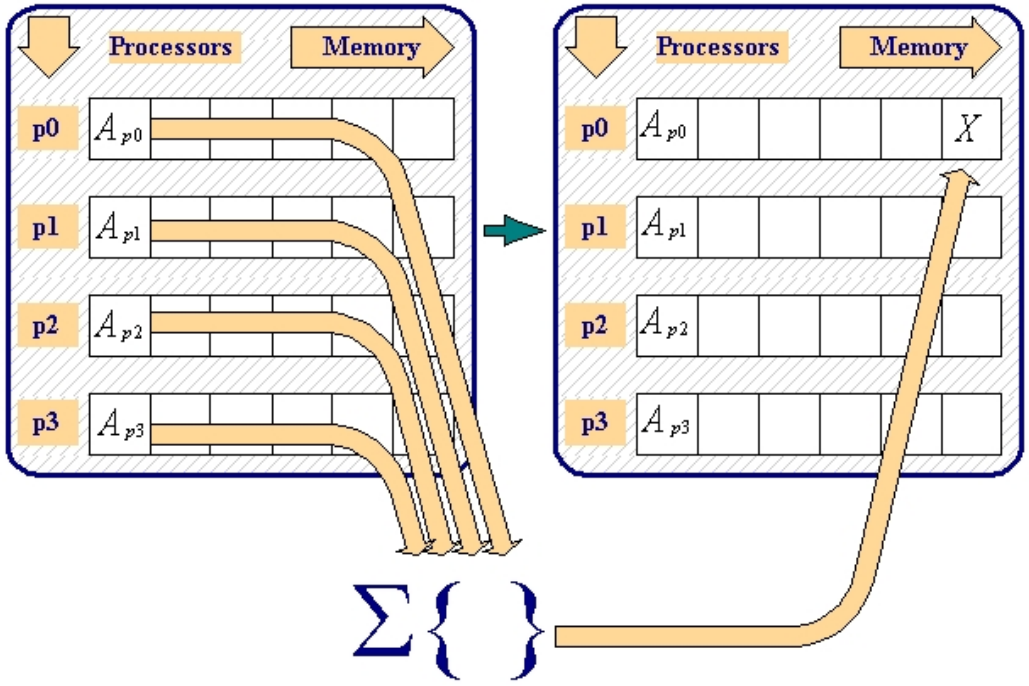
Scatter Operation




```
MPI_SCATTER(sendbuff,sendcount,sendtype,
            recvbuff,recvcount,recvtype,iroot,icomm,ierr)
```

- reverse operation of `MPI_GATHER` (source)

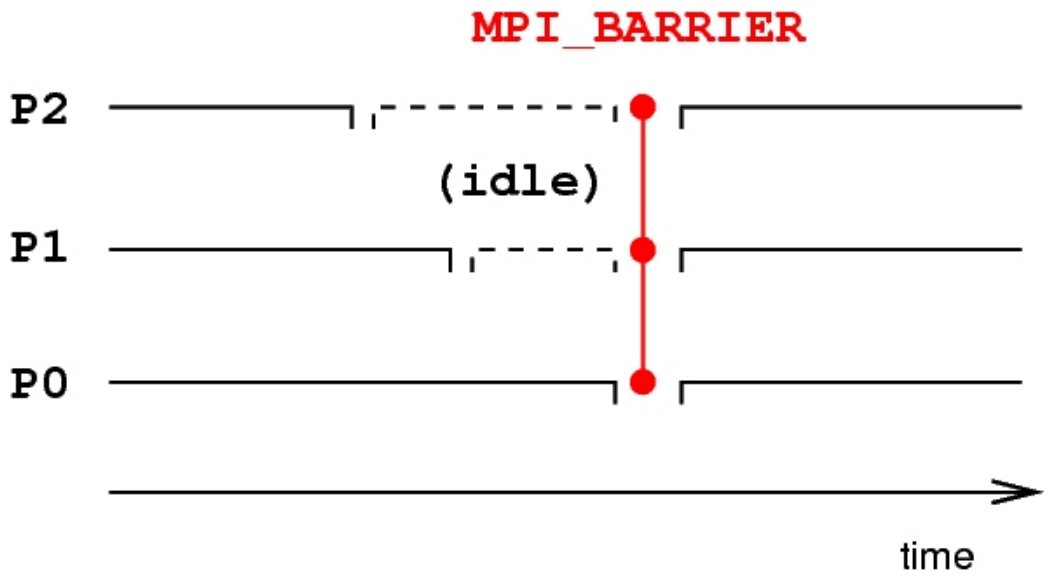
Data Reduction Operation




```
MPI_REDUCE(sendbuff,recvbuff,icount,dtype,
       operation,iroot,icomm,ierr)
```

- `operation` = `MPI_SUM,MPI_MAX,MPI_MIN,MPI_PROD,...` (source)

Barrier Synchronization



`MPI_BARRIER(icomm,ierr)`

- completion when all processes have synchronized

$\Rightarrow$ avoid if possible! (useful for debugging)

Example: Computation of π

- $\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$
- approximated by rectangular integration

1. input and broadcast of the number of intervals
2. each processor computes partial sum
3. one 'master' process collects the global result

Computation of π: Sequential

(source)

```
c************************************************************************
c   pi.f - compute pi by integrating f(x) = 4/(1 + x**2)
c   SEQUENTIAL VERSION
c
c  Variables:
c
c    pi  the calculated result
c    n   number of points of integration.
c    x           midpoint of each rectangle's interval
c    f           function to integrate
c    i           do loop index
c************************************************************************
      program main
      double precision  PI25DT
      parameter        (PI25DT = 3.14159265358979323846264338327950288d0)
      double precision  pi, h, sum, x, f, a
      integer n, i
c     /* function to integrate */
      f(a) = 4.d0 / (1.d0 + a*a)
      print *, 'SEQUENTIAL VERSION'
c     /* input of the number of intervals */
 10   write(6,98)
 98   format('Enter the number of intervals: (0 quits)')
      read(5,99) n
```

```
 99   format(i10)
c     /* check for quit signal */
      if ( n .le. 0 ) goto 30
c     /* calculate the interval size */
      h = 1.0d0/n
c     /* perform the rectangular summation */
      sum  = 0.0d0
      do 20 i = 1, n
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
 20   continue
      pi = h * sum
c     /* print result to screen */
      write(6, 97) pi, abs(pi - PI25DT)
 97   format('  pi is approximately: ', F18.16,
     +     '  Error is: ', E20.15)
c
      goto 10
 30   stop
      end
```

Computation of π: Parallel

(source)

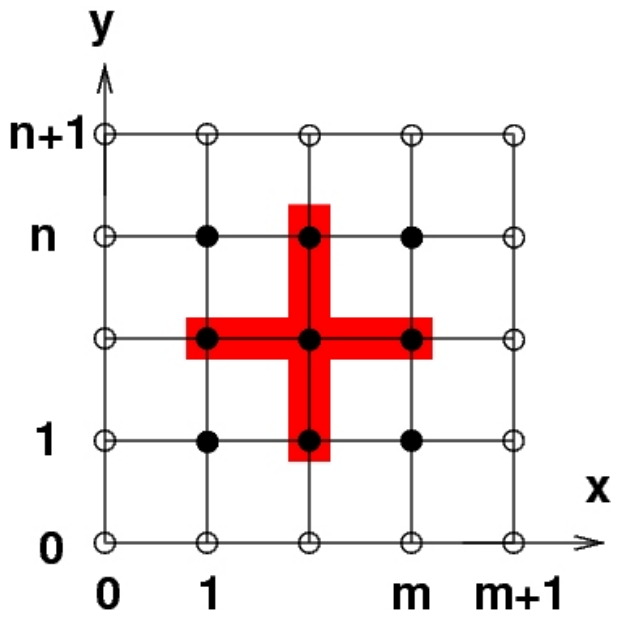
```
c     /*************************************************************/
c     /* pi.f - compute pi by integrating f(x) = 4/(1 + x**2)      */
c     /* MPI VERSION                                               */
c     /*                                                           */
c     /* Variables:                                                */
c     /*                                                           */
c     /* pi   the calculated result                                */
c     /* mypi the local sum                                        */
c     /* n    number of points of integration.                     */
c     /* x    midpoint of each rectangle's interval                */
c     /* f    function to integrate                                */
c     /* i    do loop index                                        */
c     /* numprocs total number of processors available             */
c     /* myid processor id (0<=myid<=numprocs-1)                   */
c     /*************************************************************/
      program mainpar
      include 'mpif.h'
      double precision  PI25DT
      parameter        (PI25DT = 3.14159265358979323846264338327950288d0)
      double precision  mypi,pi,h,sum,x,f,a,t0,t1,t2
      integer n,myid,numprocs,i,ierr
c     /* function to integrate */
      f(a) = 4.d0 / (1.d0 + a*a)
c     /* initialize MPI */
```

```
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)
      print *, 'Process ', myid, ' of ', numprocs, ' is alive'
c     /* input of the number of intervals */
 10   if (myid.eq.0) then
         write(6,98)
 98      format('Enter the number of intervals: (0 quits)')
         read(5,99) n
 99      format(i10)
      endif
c     /* broadcast the value for 'n' */
      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
c     /* check for quit signal */
      if ( n .le. 0 ) goto 30
c     /* start timing */
      t0=MPI_WTIME()
c     /* calculate the interval size */
      h = 1.0d0/n
c     /* perform the PARTIAL rectangular summation */
      sum  = 0.0d0
      do 20 i = myid+1, n, numprocs
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
 20   continue
      mypi = h * sum
```

```
c     /* collect all the partial sums */
      call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     $     MPI_COMM_WORLD,ierr)
c     /* stop timing & compute global maximum elapsed time*/
      t1=MPI_WTIME()
      call MPI_REDUCE(t1-t0,t2,1,MPI_DOUBLE_PRECISION,MPI_MAX,0,
     $     MPI_COMM_WORLD,ierr)
      if(myid.eq.0)write(*,*)'elapsed time: ',t2
c     /* node 0 prints result to screen */
      if (myid .eq. 0) then
         write(6, 97) pi, abs(pi - PI25DT)
 97      format('  pi is approximately: ', F18.16,
     +        '  Error is: ', E20.15)
      endif
c
      goto 10
c     /* finalize MPI */
 30   call MPI_FINALIZE(ierr)
      stop
      end
```

5. Case Study: Jacobi Iteration of 2D Poisson

$$
\begin{aligned}
(\partial_{xx} + \partial_{xx})\, u &= 0 \\
u(\mathbf{x} \in \Gamma) &= u_\Gamma
\end{aligned}
$$



- 2nd order central finite-differences, uniform grid:

$$
u_{i,j}^{n+1} = \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4}
$$

Sequential Code

(source)

```
      program jacobi
      implicit real*8 (a-h,o-z)
c     /*************************************************************/
c     /* jacobi iteration for poisson problem in 2d, discretized by  */
c     /* second-order central finite-difference on cartesian mesh.   */
c     /* single processor version.                                   */
c     /* m.u. 26/10/2004                                             */
c     /*************************************************************/
c     /* (global) problem size, number of processors, type of solution */
      parameter(m=10,n=10,nproc=1,isol=2)
c
c     /* max. number of iterations allowed & tolerance*/
      parameter(nitmx=100000,tol=1.e-5)
c
c     /* arrays */
      dimension aloc(0:m+1,0:n+1),adebug(0:m+1,0:n+1)
c
c     /* other stuff */
      logical lconverged
      external lconverged
c-----|--------------------------------------------------------------|
c     /* initialize the physical boundaries & set interior to zero*/
      call assign(aloc,m,n,isol)
```

```
      call write_array(aloc,m,n,isol,55)
c
      do 100 iter=1,nitmx       !this is the main iteration loop
c
         call jacobi_step(aloc,adebug,m,n,diffl,valmxl)
c
         if(lconverged(diffl,valmxl,tol))goto 110
c
 100  continue
      write(*,*)'jacobi iteration did not converge! ',iter,
     $     diffl/valmx,tol
 110  continue
      write(*,*)'jacobi iteration converged in ',iter,' steps.'
c
c     /* compute the error of the solution */
      call check_array(aloc,m,n,isol)
c
c     /* write result to file (along with solution) */
      call write_array(aloc,m,n,isol,44)
c
      stop
      end
c----+-------------------------------------------------------------|
      subroutine jacobi_step(a,tmp,m,n,diff,valmx)
      implicit real*8 (a-h,o-z)
c
```

```
c     /*****************************************************************/
c     /* performs one step of jacobi iteration for 2d matrix in 1d   */
c     /* cartesian domain decomposition along 2nd index.             */
c     /*                                                             */
c     /* input:                                                      */
c     /* a(0:m+1,0:n+1)          matrix in local memory              */
c     /* tmp(0:m+1,0:n+1)        work array                          */
c     /* my_rank                 processor id                        */
c     /*                                                             */
c     /* output:                                                     */
c     /* a(0:m+1,0:n+1)          updated matrix                      */
c     /* diff                    the r.m.s. of the increment         */
c     /*                         relative to the maximum value of the*/
c     /*                         function (for convergence measure)  */
c     /*****************************************************************/
c
      dimension a(0:m+1,0:n+1),tmp(0:m+1,0:n+1)
c----+-----------------------------------------------------------------|
      pi=4.*atan(1.)
c
      if=1
      il=m
      jf=1
      jl=n
      diff=0.0
      valmx=0.0
```

```
      do j=jf,jl
         do i=if,il
            tmp(i,j)=(a(i,j+1)+a(i+1,j)+a(i-1,j)+a(i,j-1))*.25d0
            diff=max(diff,abs(tmp(i,j)-a(i,j)))
            valmx=max(valmx,abs(tmp(i,j)))
         enddo
      enddo
      do j=jf,jl
         do i=if,il
            a(i,j)=tmp(i,j)
         enddo
      enddo
c
      return
      end subroutine jacobi_step
c-----+--------------------------------------------------------------|
      logical function lconverged(diffl,valmxl,tol)
      implicit real*8 (a-h,o-z)
c
      lconverged=.false.
c
      diffl=diffl/valmxl
      if(diffl.le.tol)then
         lconverged=.true.
         write(*,*)' we have converged!',diffl,tol
      endif
```

```
c
      return
      end function lconverged
c----+-----------------------------------------------------------------|
      subroutine assign(a,m,n,itype)
      implicit real*8 (a-h,o-z)
c
c     /*****************************************************************/
c     /* initializes the matrix a to some values.                      */
c     /*                                                               */
c     /* input:                                                        */
c     /* a(m,n)                  matrix in local memory                */
c     /* itype                   type of the requested values.         */
c     /*                         =1: a(i,j)=i*j                        */
c     /*                         =2: a(i,j)=sin(pi*xi)*exp(-pi*yj)     */
c     /*                                                               */
c     /* output:                                                       */
c     /* a(m,n)                  newly assigned matrix                 */
c     /*****************************************************************/
      dimension a(0:m+1,0:n+1)
c----+-----------------------------------------------------------------|
      pi=4.*atan(1.)
c
      if=1
      il=m
      jf=1
```

```
      jl=n
      if(itype.eq.2)then
c     /* initialize the interior of the matrix to zero */
         do j=jf,jl
            do i=if,il
               a(i,j)=0.0       !all to zero
            enddo
         enddo
c     /* set physical boundaries:   */
c     /* aij=sin(pi*xi)*exp(-pi*yj) with xi=i/(m+1) \forall 0<=i<=m+1 */
c     /*                            with yj=j/(n+1) \forall 0<=j<=n+1 */
         do j=jf-1,jl+1
            yy=float(j)/float(n+1)
            i=if-1
            xx=float(i)/float(m+1)
            a(i,j)=solution(xx,yy,itype)
            i=il+1
            xx=float(i)/float(m+1)
            a(i,j)=solution(xx,yy,itype)
         enddo
         j=jf-1
         yy=float(j)/float(n+1)
         do i=if-1,il+1
            xx=float(i)/float(m+1)
            a(i,j)=solution(xx,yy,itype)
         enddo
```

```
         j=jl+1
         yy=float(j)/float(n+1)
         do i=if-1,il+1
            xx=float(i)/float(m+1)
            a(i,j)=solution(xx,yy,itype)
         enddo
      else
         write(*,*)'assign: this type not implemented: ',itype
         stop
      endif
c
      return
      end subroutine assign
c----+-----------------------------------------------------------------|
      subroutine check_array(a,m,n,itype)
      implicit real*8 (a-h,o-z)
c
c     /*****************************************************************/
c     /* computes the error w.r.t. to analytic solution               */
c     /*                                                               */
c     /* input:                                                        */
c     /* a(m,n)                  matrix in local memory                */
c     /* itype                   type of the requested values.         */
c     /*                         =1: a(i,j)=i*j                        */
c     /*                         =2: a(i,j)=sin(pi*xi)*exp(-pi*yj)     */
c     /*                                                               */
```

```
c     /* output:                                                        */
c     /*****************************************************************/
      dimension a(0:m+1,0:n+1)
c----+----------------------------------------------------------------|
      pi=4.*atan(1.)
c
      if=1
      il=m
      jf=1
      jl=n
      errmx=0.d0
      valmx=0.d0
      if(itype.eq.2)then
         do j=jf,jl
            yy=float(j)/float(n+1)
            do i=if,il
               xx=float(i)/float(m+1)
               sol=solution(xx,yy,itype)
               errmx=max(errmx,abs(a(i,j)-sol))
               valmx=max(valmx,sol)
            enddo
         enddo
         write(*,*)'maximum relative error: ',errmx/valmx,errmx,valmx
      endif
c
      return
```

```
      end subroutine check_array
c-----+-----------------------------------------------------------------|
      subroutine write_array(a,m,n,itype,iunit)
      implicit real*8 (a-h,o-z)
c
c     /*****************************************************************/
c     /* writes 2d array to file, in ascii format with line skip     */
c     /* for gnuplot                                                 */
c     /*                                                             */
c     /* input:                                                      */
c     /* a(m,n)                  matrix in local memory              */
c     /* itype                   type of the requested values.       */
c     /*                         =1: a(i,j)=i*j                      */
c     /*                         =2: a(i,j)=sin(pi*xi)*exp(-pi*yj)   */
c     /* iunit                   unit number for output              */
c     /*                                                             */
c     /* output:                                                     */
c     /*****************************************************************/
      dimension a(0:m+1,0:n+1)
c-----+-----------------------------------------------------------------|
      do j=0,n+1
         yy=float(j)/float(n+1)
         do i=0,m+1
            xx=float(i)/float(m+1)
            sol=solution(xx,yy,itype)
            write(iunit,112)i,j,xx,yy,a(i,j),sol,a(i,j)-sol
```

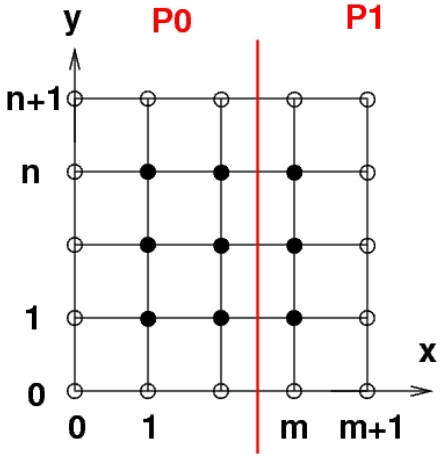
```
         enddo
         write(iunit,111)
      enddo
 111  format(' ')
 112  format(1x,2(i5,1x),10(e15.8,1x))
c
      return
      end subroutine write_array
c----+-----------------------------------------------------------------|
      real*8 function solution(x,y,itype)
      implicit real*8 (a-h,o-z)
      pi=4.*atan(1.)
c
      if(itype.eq.2)then
         solution=sin(pi*x)*exp(-pi*y)
      else
         write(*,*)'solution: this type not implmented! ',itype
         stop
      endif
c
      return
      end function solution
c----+-----------------------------------------------------------------|
```

Parallel Code

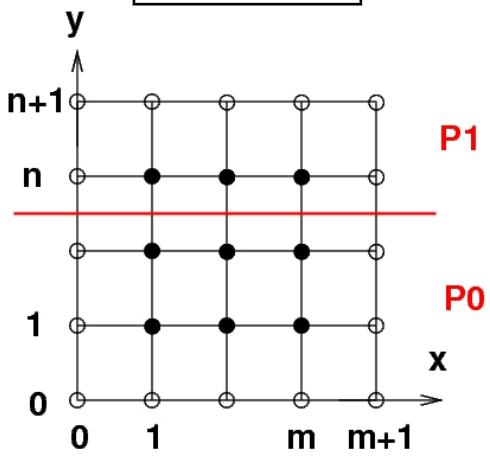
1. domain/data decomposition
2. design of communication patterns
3. implementation
4. validation against sequential results
5. timing/optimization of parallel execution

Domain Decomposition

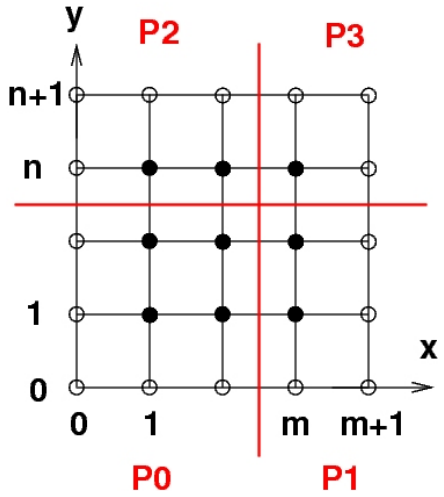
1D in x



1D in y



2D



	1D in x	1D in y	2D
messages:	$(\mathtt{np}-1)2$	$(\mathtt{np}-1)2$	$(\mathtt{nxp}-1)2\,\mathtt{nyp}$ $+\,(\mathtt{nyp}-1)2\,\mathtt{nxp}$
size:	$(\mathtt{np}-1)2n$	$(\mathtt{np}-1)2m$	$(\mathtt{nxp}-1)2\,n$ $+\,(\mathtt{nyp}-1)2\,m$

- for large np: 2D decomposition $\Rightarrow$ more messages, smaller total volume

Domain/Data Decomposition

- 2D decomposition is most flexible
- BUT: for simplicity we choose 1D decomposition
- no decomposition for first array index

  $\longrightarrow$ contiguous data (FORTRAN)

1D Decomposition in Y-Direction: Pointers

relation between global and local indices

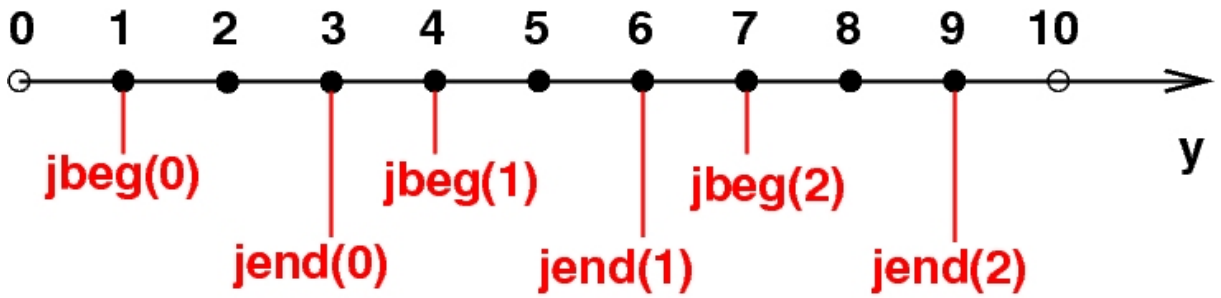
$\Rightarrow$ define pointer arrays (known to all processes):

```
integer jbeg(0:nproc-1)
integer jend(0:nproc-1)
```

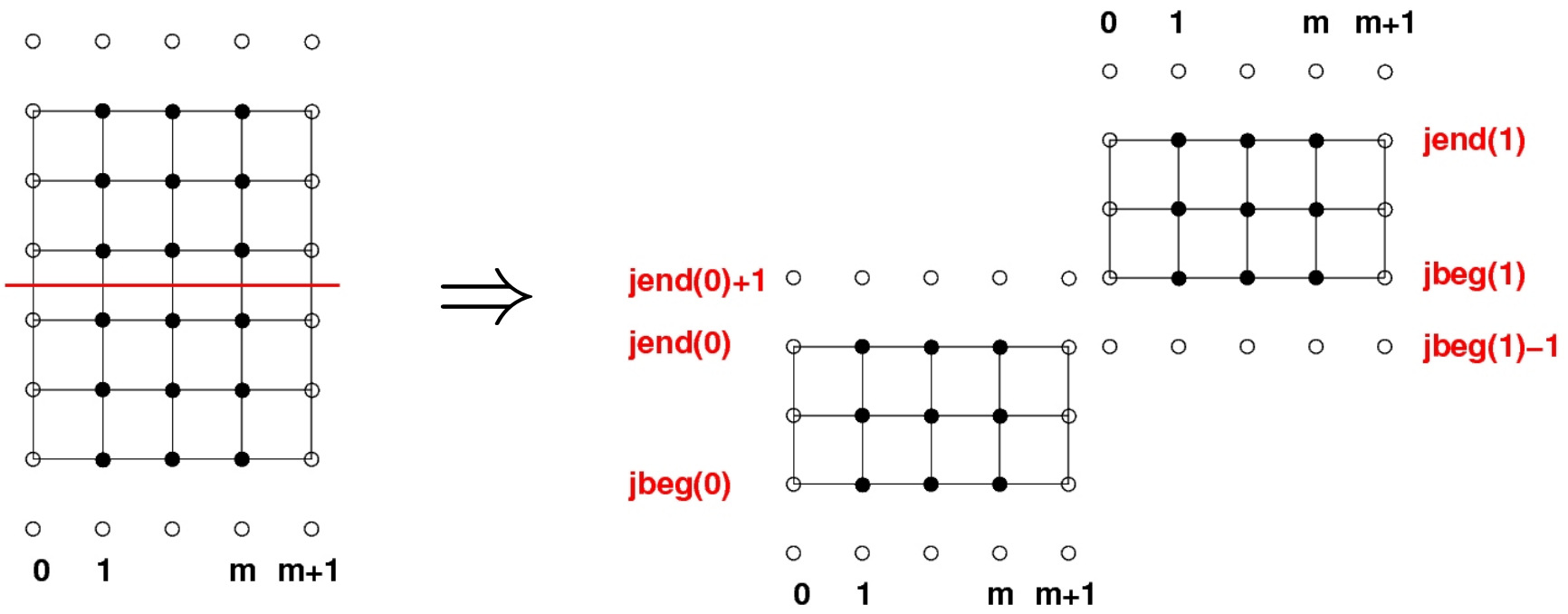
example:

nproc=3

n=9



Ghost Cells



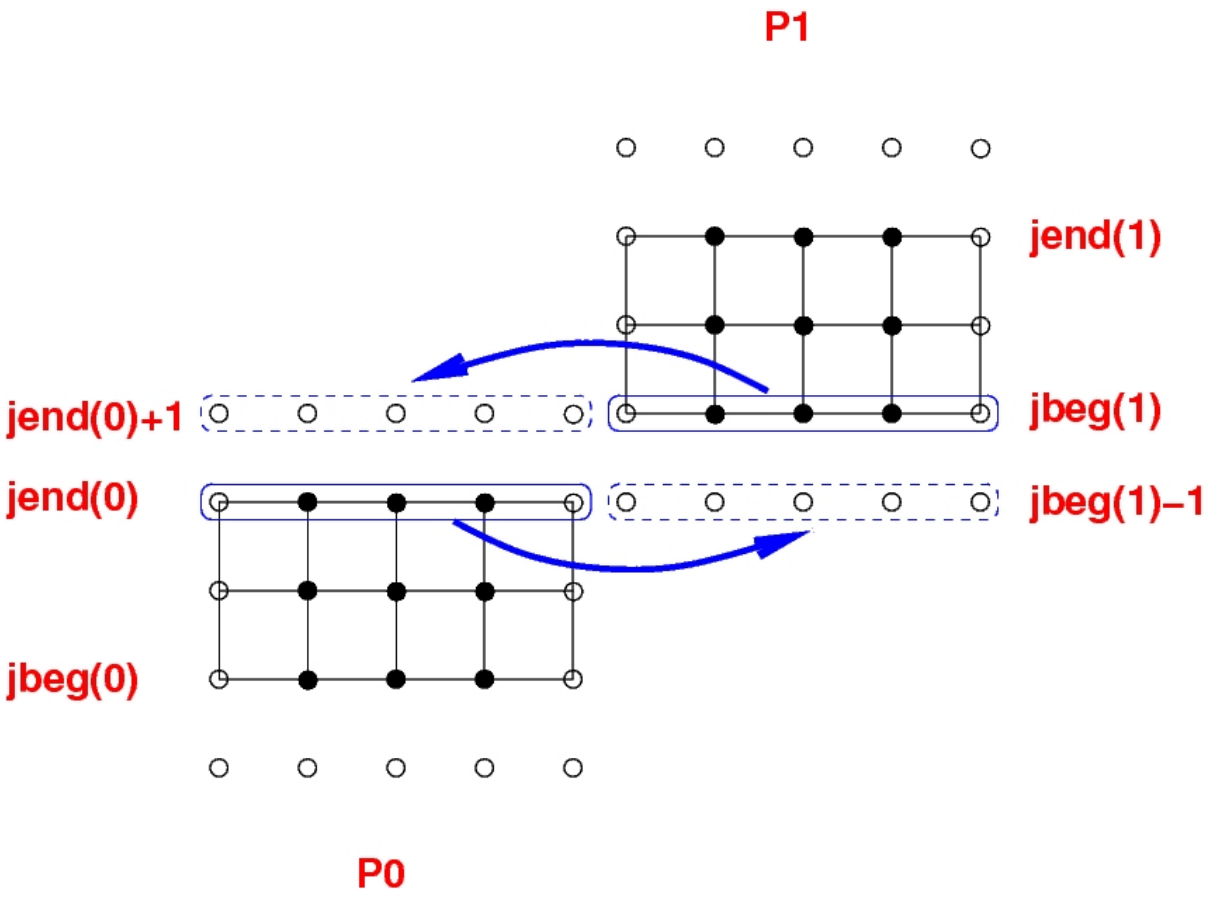
⇒ arrays have size: `A(0:m+1,jbeg(myrank)-1:jend(myrank)+1)`

Pointer Routine

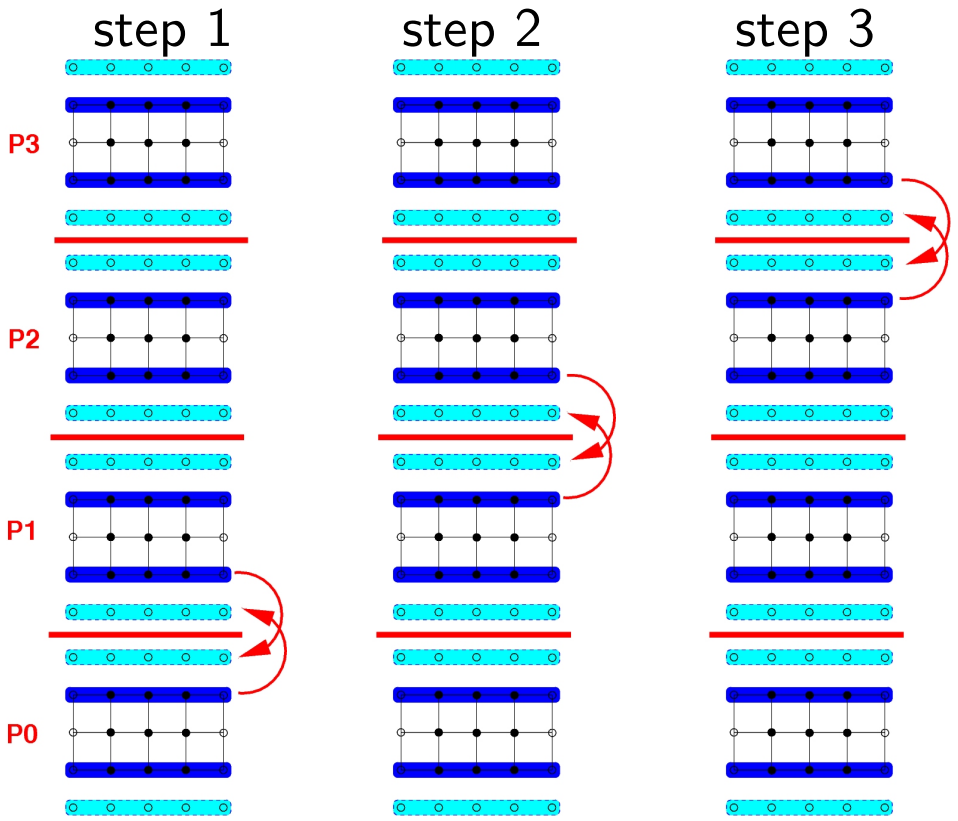
(source)

```
      subroutine MPE_DECOMP1D_all(n,numprocs,s,e)
      implicit none
c     /* input                                                          */
c     /* n             data length to be decomposed (1:n)               */
c     /* numprocs      no. of processors                                */
c     /* output                                                         */
c     /* s(0:numprocs-1) pointer to start of memory local to procs      */
c     /* e(0:numprocs-1) pointer to end of memory local to procs        */
      integer n, numprocs, myid, s(0:numprocs-1), e(0:numprocs-1)
      integer nlocal,deficit
c---- +--------------------------------------------------------------|
      do myid=0,numprocs-1
         nlocal  = n / numprocs
         s(myid) = myid * nlocal + 1
         deficit = mod(n,numprocs)
         s(myid) = s(myid) + min(myid,deficit)
         if (myid .lt. deficit) then
            nlocal = nlocal + 1
         endif
         e(myid) = s(myid) + nlocal - 1
         if (e(myid) .gt. n .or. myid .eq. numprocs-1) e(myid) = n
      enddo
      return
      end
```

Ghost Cells: Data Exchange



Communication Pattern 1



(source)

blocking communication

- 'domino' effect:

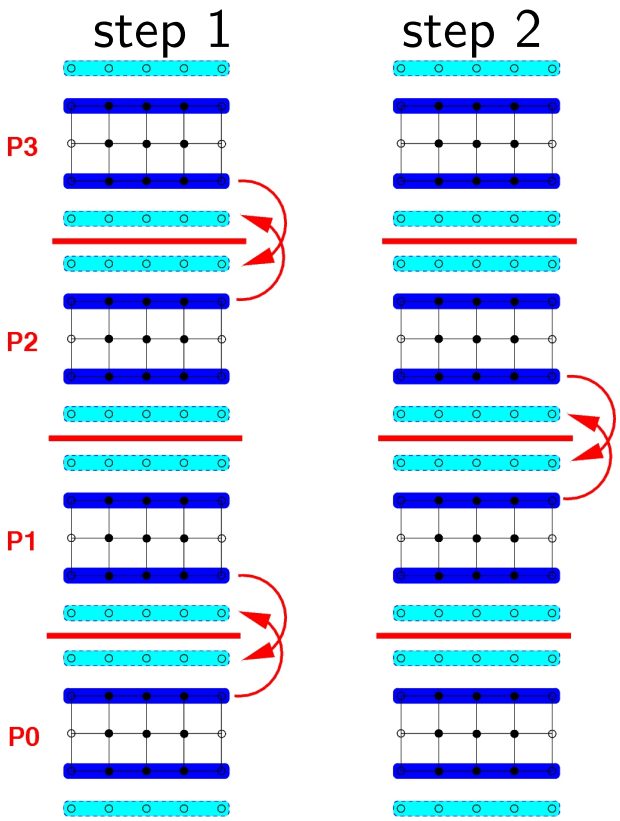
$\Rightarrow$ needs `nproc`-1 steps

Communication Pattern 2

non-blocking send/receive:

- each processor posts 2 send and 2 receive operations
- MPI internally decides the order of the communication
- all processors wait for completion before proceeding

Communication Pattern 4



blocking communication

- alternating pairs communicate

$\Rightarrow$ needs only 2 steps

Non-Blocking Communication Routine

(source)

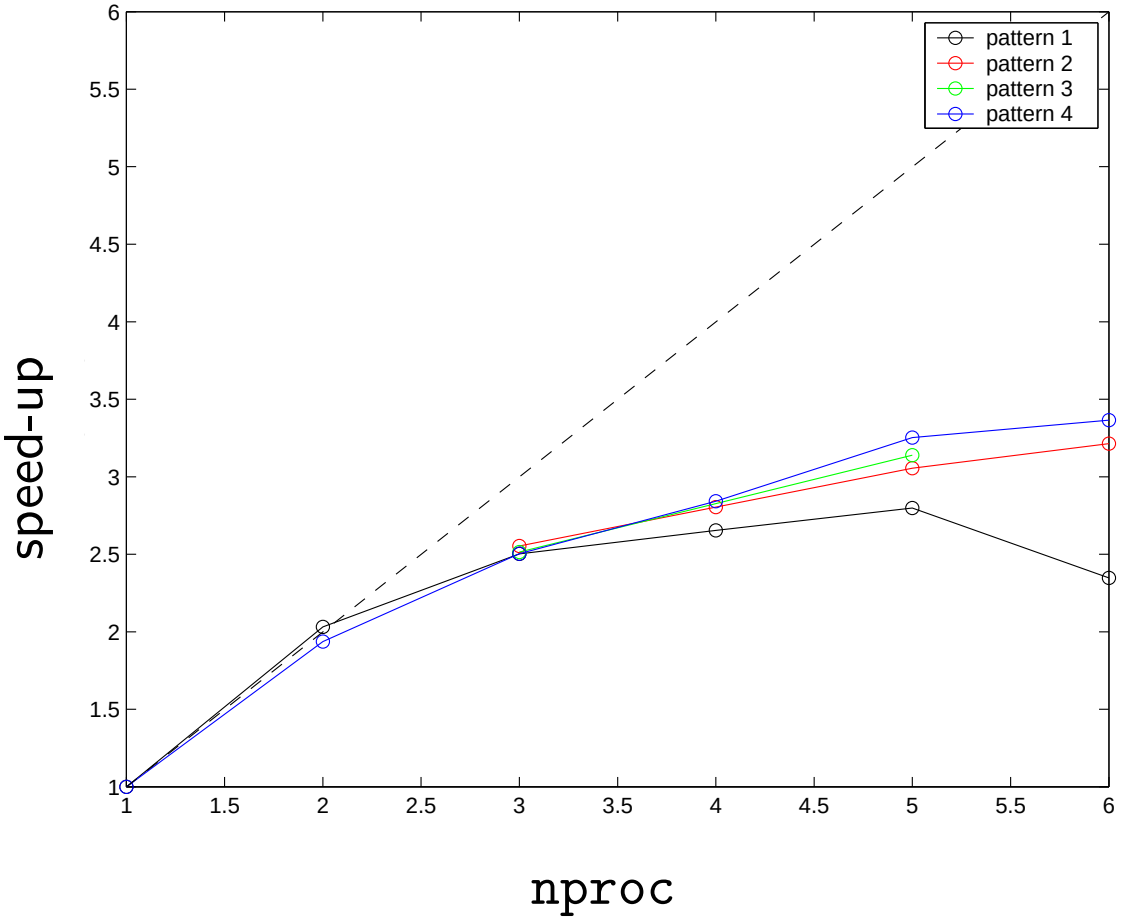
```
      subroutine exchange_ghostline2(a,m,jf,jl,my_rank,nproc,mpi_comm,irealtype)
#include "realdef.inc"
      include "mpif.h"
c     /* performs the communication of ghost lines on both extremes of */
c     /* the second index of a 2d field between neighboring processors */
c     /* in a 1d cartesian processor "grid". extreme processors do not */
c     /* exchange as their ghost lines correspond to physical bcs.     */
c     /*                                                               */
c     /* uses MPI_ISEND, MPI_IRECV therefore the communication is done */
c     /* non-blockingly in 4 steps.                                    */
c     /*                                                               */
c     /* input                                                         */
c     /* a(0:m+1,jf-1:jl+1)   array in local memory                    */
c     /* my_rank              processor rank                           */
c     /* nproc                no of procs in communicator              */
c     /* mpi_comm             mpi communicator                         */
c     /* irealtype            the data-type for the current floats     */
c     /*                                                               */
c     /* output                                                        */
c     /* a(0:m+1,jf-1:jl+1)   array in local memory; the ghost lines   */
c     /*                       have been exchanged with neighbors,     */
c     /*                       except for extreme processors           */
c     /* mpi related */
```

```
      parameter(nmsg=2)
      integer istat(MPI_STATUS_SIZE,2*nmsg),ireq(2*nmsg)
      integer iseq_send(nmsg),iseq_recv(nmsg)
      data iseq_send /-1,+1/ !order of the send operations
      data iseq_recv /+1,-1/ !order of the recv operations
c     /* arguments */
      dimension a(0:m+1,jf-1:jl+1)
c     /* etc */
      parameter(itag=10)
c-----|--------------------------------------------------------------|
      nline=m+2
      nreq=0
c
      do ii=1,nmsg
c     /* send to next in sender-list 'iseq_send' */
         idest=my_rank+iseq_send(ii)
         if (idest.ge.0.and.idest.le.nproc-1) then
            if(iseq_send(ii).eq.+1)then
               jj=jl
            else
               jj=jf
            endif
            nreq=nreq+1
#ifdef DEBUG_COMM
      write(*,*)my_rank,' about to MPI_ISEND ',nline,jj,' to ',idest
#endif
```

```
            call MPI_ISEND(a(0,jj),nline,irealtype,
     $           idest,itag,mpi_comm,ireq(nreq),ierr)
#ifdef DEBUG_COMM
      write(*,*)my_rank,' finished MPI_ISEND ',nline,jj,' to ',idest
#endif
         end if
c
c     /* receive from next in receiver-list 'iseq_recv' */
         iorig=my_rank+iseq_recv(ii)
         if (iorig.ge.0.and.iorig.le.nproc-1) then
            if(iseq_recv(ii).eq.+1)then
               jj=jl+1
            else
               jj=jf-1
            endif
            nreq=nreq+1
#ifdef DEBUG_COMM
      write(*,*)my_rank,' about to MPI_IRECV ',nline,jj,' from ',iorig
#endif
            call MPI_IRECV(a(0,jj),nline,irealtype,
     $           iorig,itag,mpi_comm,ireq(nreq),ierr)
#ifdef DEBUG_COMM
      write(*,*)my_rank,' finished MPI_IRECV ',nline,jj,' from ',iorig
#endif
         end if
      enddo
```

```
c
c     /* wait for all the posted operations to complete */
#ifdef DEBUG_COMM
      write(*,*)my_rank,' waiting at MPI_WAITALL ',nreq
#endif
      call MPI_WAITALL(nreq,ireq,istat,ierr)
#ifdef DEBUG_COMM
      write(*,*)my_rank,' finished MPI_WAITALL ',nreq
#endif
      return
      end
```

Parallel Speed-Up



problem size:

n=m=250

(SGI Itanium)

6. Extensions

- non-contiguous data: MPI derived data types
- MPI virtual topologies: communication sub-groups
- MPI parallel I/O
- additional parallel mathematical libraries (ScaLAPACK)
- ...

more information: Web Course, MPI Standard, MPI Reference