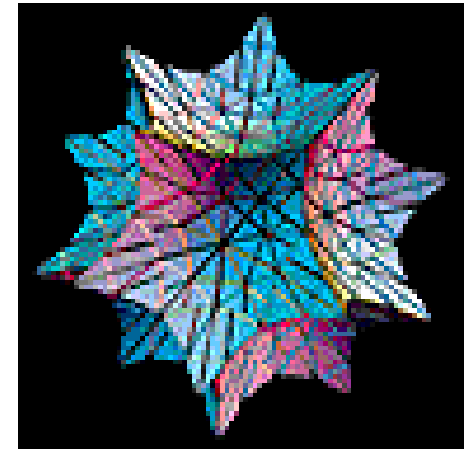
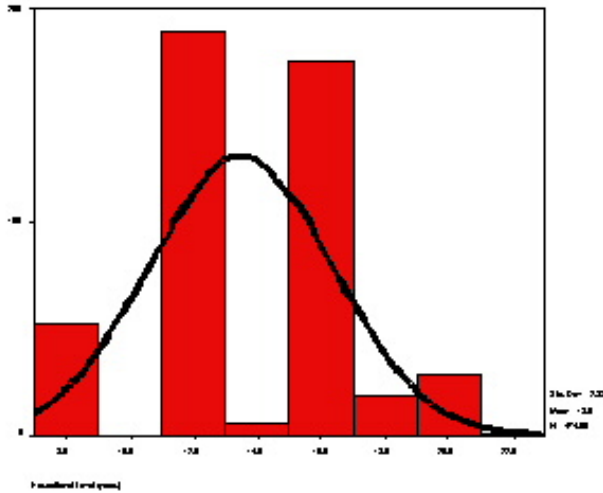


Implementing Parallel Codes

February 15, 2006



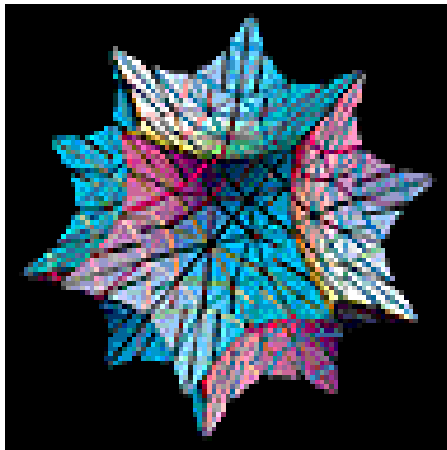
Presented by the

ITC Research Computing Support Group

Kathy Gerber, Ed Hall, Katherine Holcomb, Nancy Kechner, Tim F. Jost Tolson

- 3/15 Carla Lee, Brown Science and Engineering Library: topic TBA
- 3/29 Ed Hall: pMatlab: Parallel Matlab Toolbox
- 4/12 Chris Gist (Alderman Library): Image Mapper: Easy mapping on the Web
- 4/26 Kathy Gerber: Modern Classification Methods

ITC Research Computing Support Implementing Parallel Codes



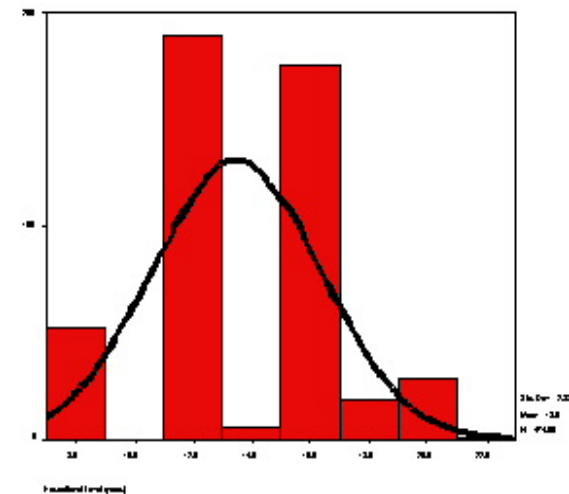
By: Katherine Holcomb

Research Computing Support Center

Phone: 243-8800 Fax: 243-8765

E-Mail: Res-Consult@Virginia.EDU

<http://www.itc.Virginia.edu/researchers>



MPI=Message-Passing Interface

- Standard
 - Defined by an open specification
- Portable
 - Implemented by most vendors of MPP and clusters
- Simple
 - Basic functionality is easy to learn

MPI is a Library

- Standard bindings exist for C, C++, and Fortran
 - C++ programmers can also use the C bindings (often this is simpler)
- Fortran 90 users can use a module
 - use mpi
 - or can include the Fortran 77 include file

Messages

In MPI, a message consists of data+envelope

The envelope is information that uniquely identifies the source, the destination, and the identification of the message. It consists of

Sender's rank:
the process ID

Receiver's rank:
the process ID

Tag:
an arbitrary identifier

Communicator:
an ID for a group of processes that can exchange data

MPI supplies a predefined communicator, `MPI_COMM_WORLD`, consisting of all processes running at the start of execution

MPI Datatype

- **MPI_INT**
- **MPI_SHORT**
- **MPI_LONG**
- **MPI_CHAR**
- **MPI_UNSIGNED_CHAR**
- **MPI_UNSIGNED_SHORT**
- **MPI_UNSIGNED**
- **MPI_FLOAT**
- **MPI_DOUBLE**
- **MPI_LONG_DOUBLE**
- **MPI_BYTE**
- **MPI_PACKED**

C

- signed int
- signed short
- signed long
- signed char
- unsigned char
- unsigned short
- float
- double
- long double

MPI Datatype

- **MPI_INTEGER**
- **MPI_REAL**
- **MPI_DOUBLE_PRECISION**
- **MPI_COMPLEX**
- **MPI_LOGICAL**
- **MPI_CHARACTER**
- **MPI_BYTE**
- **MPI_PACKED**

Fortran

- **INTEGER**
- **REAL**
- **DOUBLE PRECISION**
- **COMPLEX**
- **LOGICAL**
- **CHARACTER**

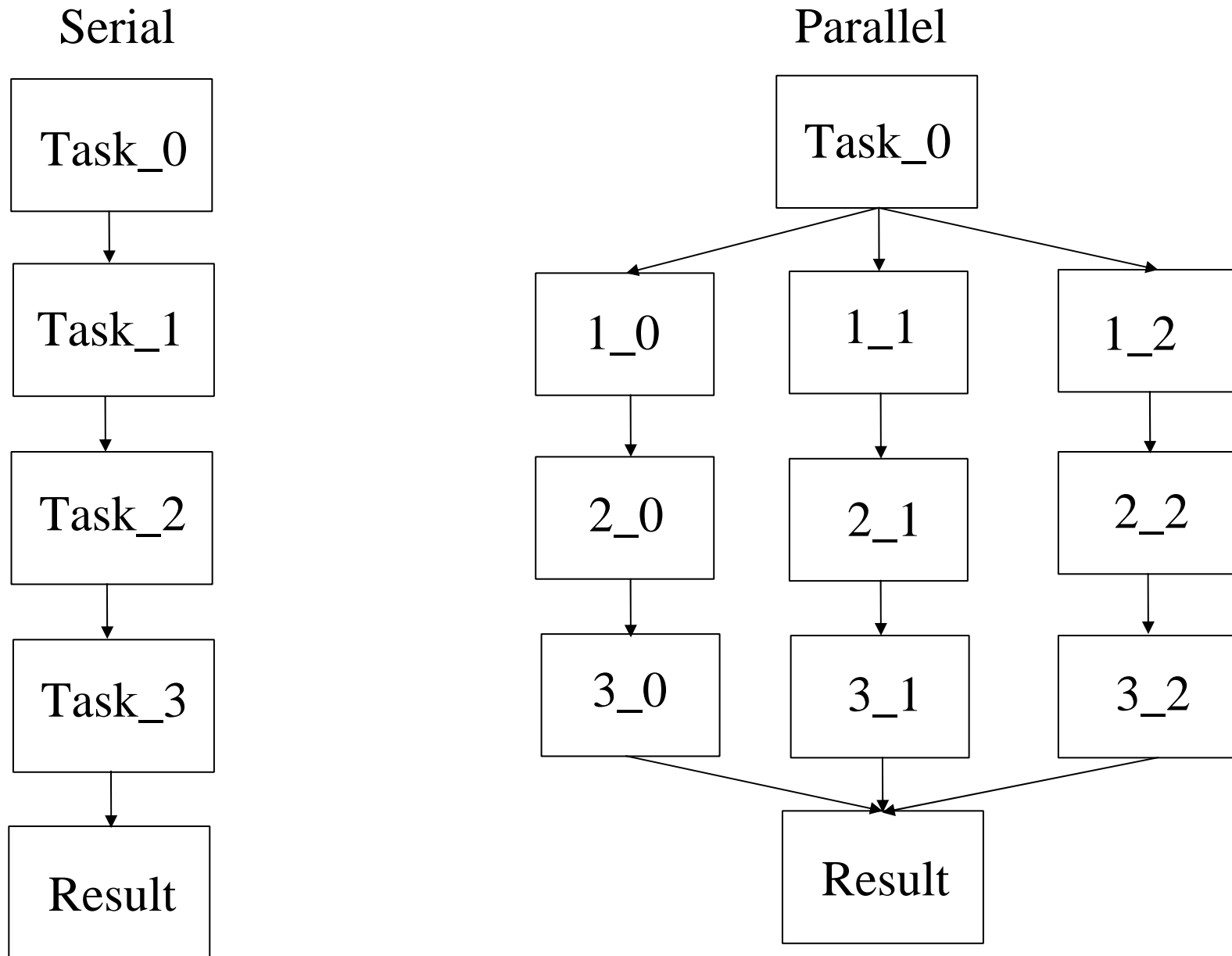
Collective Communications

- Scatter/Broadcast: send information from one processor to all. Broadcast sends same data to all; scatter can send different data to each process.
- Gather: receive information from all processes at one process.
- Barrier Synchronization
- Global Reduction Operations
 - Sum, product, max, min, others: gathers data and performs global operation

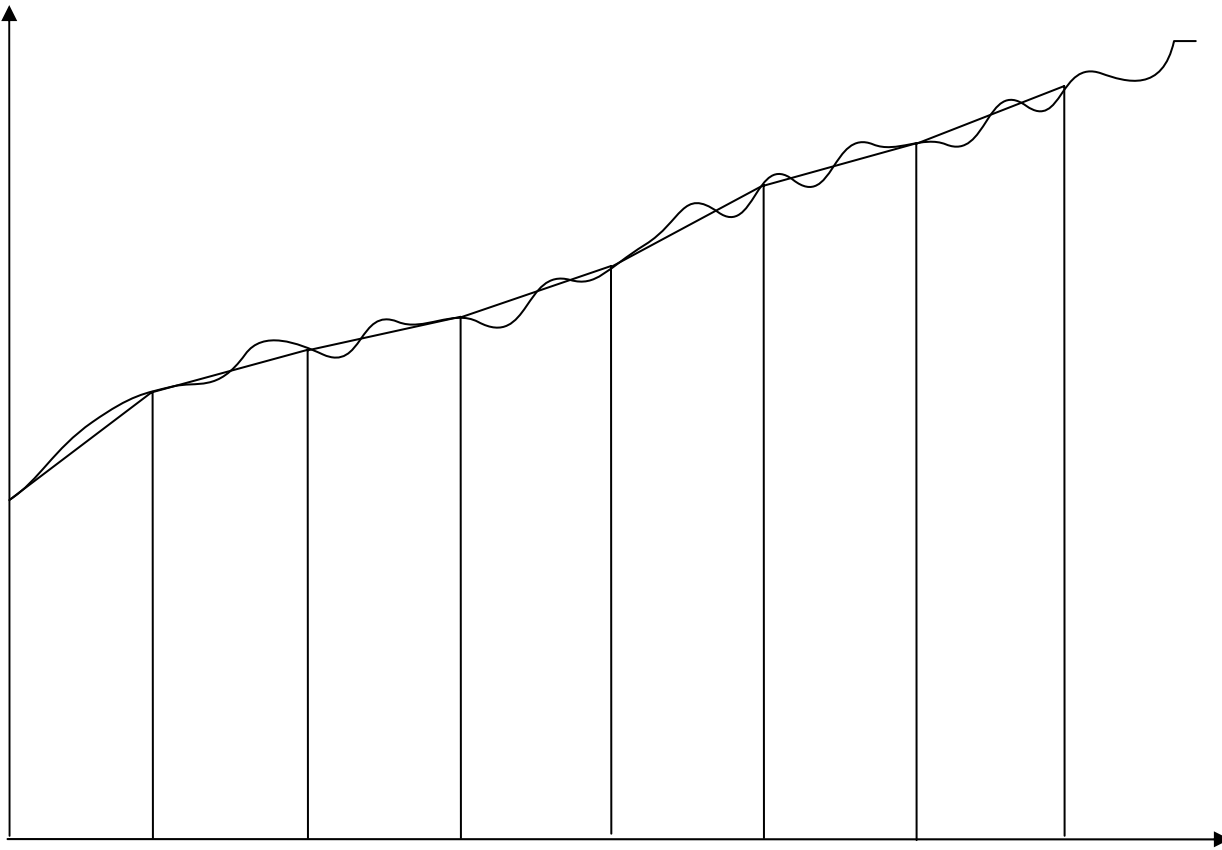
Global Reduction Operations

- MPI_MAX maximum
- MPI_MIN minimum
- MPI_SUM sum
- MPI_PROD product
- MPI_I_AND logical and
- MPI_BAND bitwise and
- MPI_I_OR logical or
- MPI_BOR bitwise or
- MPI_I_XOR logical xor
- MPI_BXOR bitwise xor

Collective Communications



EXAMPLE 1: Parallelizing the Trapezoid Rule



```
program trapezoid
implicit none
```

```
! Calculate a definite integral using trapezoid rule
```

```
real      :: a, b
integer   :: n
```

```
real      :: h, integral
```

```
real      :: f,x
integer   :: i
```

```
read(*,*) a, b, n
h=(b-a)/n
```

```
integral = (f(a) + f(b))/2.0
```

```
x=a
```

```
do i=1, n-1
```

```
    x = x+h
```

```
    integral = integral + f(x)
```

```
enddo
```

```
integral = h*integral
```

```
print *, integral
```

```
stop
```

```
end
```

```
program partrap
implicit none

real    :: integral, total
real    :: a, b, h
integer:: n

real    :: local_a, local_b
integer:: local_n

real    :: trap, f

include 'mpif.h'

integer:: my_rank, p
integer, parameter :: tag=50
integer:: ierr, status(MPI_STATUS_SIZE)

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

if (my_rank .eq. 0) then
    read(*,*) a, b, n
endif
```

```
call MPI_Bcast(a,1,MPI_REAL,0,MPI_COMM_WORLD,ierr)
call MPI_Bcast(b,1,MPI_REAL,0,MPI_COMM_WORLD,ierr)
call MPI_Bcast(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

h= (b-a)/n
local_n = n/p

local_a = a + my_rank*local_n*h
local_b = local_a + local_n*h
integral = trap(local_a, local_b, local_n, h)

call MPI_Reduce(integral, total, 1, MPI_REAL, MPI_SUM, &
                0, MPI_COMM_WORLD,ierr)

if (my_rank .eq. 0) then
    print *, total
endif

call MPI_Finalize(ierr)

stop
end
```

```

real function trap(local_a, local_b, local_n, h)

implicit none

real    :: local_a, local_b, h
integer:: local_n

real    :: f,x
integer:: i
real    :: parint

parint = (f(local_a) + f(local_b))/2.0

trap    = parint*h

return
end

real function f(x)
implicit none
real    :: x

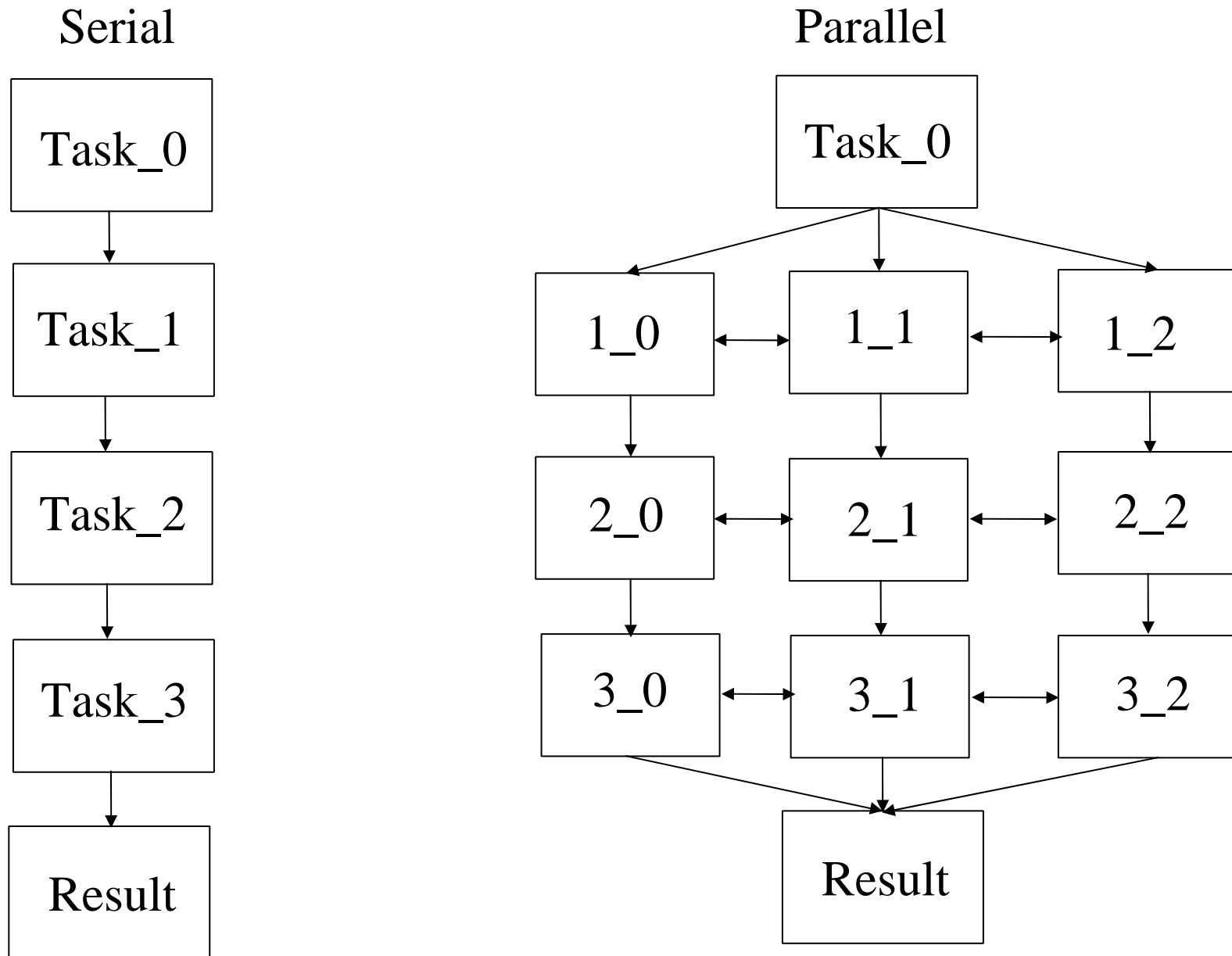
f=sin(x) + cos(x)
return
end

```

Point-to-Point Communications

- Send and Receive
 - Process sends data from itself to one other process or receives data from one other process
 - Accomplished via writing and reading buffers
 - Blocking or Nonblocking
- Within a processor, messages are *ordered*.
 - First message sent arrives first, second next, etc.
- Among different processes, messages are **not** ordered.
 - Parallel codes are, in general, nondeterministic.

A Common Parallel Pattern



MPI Send-Receive

Blocking send and receive

Send: returns when message has been buffered or sent so that memory allocated for the message can be reused – does not mean delivered!

Receive: returns when data has been received into memory referenced for data

C:

```
int my_rank, my_neighbor, tag=50;
```

```
MPI_Status status
```

```
int mdata[100], idata[100]
```

```
...
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)
```

```
...
```

```
MPI_Send(mdata,100,MPI_INT,my_neighbor,tag, MPI_COMM_WORLD);
```

```
MPI_Recv(idata,100,MPI_INT,my_neighbor,tag,MPI_COMM_WORLD,&status);
```

Blocking Send-Recv and Safety

Deadlock:

```
call MPI_Comm_rank(comm,rank,ierr)
if (rank .eq. 0) then
    call MPI_Recv(recvbuf,count,MPI_REAL,1,tag,comm,status,ierr)
    call MPI_Send(sendbuf,count,MPI_REAL,1,tag,comm,ierr)
else if (rank .eq. 1) then
    call MPI_Recv(recvbuf,count,MPI_REAL,0,tag,comm,status,ierr)
    call MPI_Send(sendbuf,count,MPI_REAL,0,tag,comm,ierr)
endif
```

Safe:

```
call MPI_Comm_rank(comm,rank,ierr)
if (rank .eq. 0) then
    call MPI_Send(sendbuf,count,MPI_REAL,1,tag,comm,ierr)
    call MPI_Recv(recvbuf,count,MPI_REAL,1,tag,comm,status,ierr)
else if (rank .eq. 1) then
    call MPI_Recv(recvbuf,count,MPI_REAL,0,tag,comm,status,ierr)
    call MPI_Send(sendbuf,count,MPI_REAL,0,tag,comm,ierr)
endif
```

Another Unsafe Pattern

Unsafe: Sending to two neighbors simultaneously

```
call MPI_Comm_rank(comm,rank,ierr)
```

```
call MPI_Send(rightbuf,count,MPI_REAL,rank+1,tag,comm,ierr)
```

```
call MPI_Send(leftbuf,count,MPI_REAL,rank-1,tag,comm,ierr)
```

```
call MPI_Recv(leftbc,count,MPI_REAL,rank-1,tag,comm,status,ierr)
```

```
call MPI_Recv(rightbc,count,MPI_REAL,rank+1,tag,comm,ierr)
```

Safe:

```
call MPI_Comm_rank(comm,rank,ierr)
```

```
if (mod(rank,2) .ne. 0) then
```

```
    call MPI_Send(rightbuf,count,MPI_REAL,rank+1,tag,comm,ierr)
```

```
    call MPI_Recv(leftbc,count,MPI_REAL,rank-1,tag,comm,status,ierr)
```

```
else if (rank .eq. 1) then
```

```
    call MPI_Recv(rightbuf,count,MPI_REAL,rank+1,tag,comm,status,ierr)
```

```
    call MPI_Send(leftbc,count,MPI_REAL,rank-1,tag,comm,ierr)
```

```
endif
```

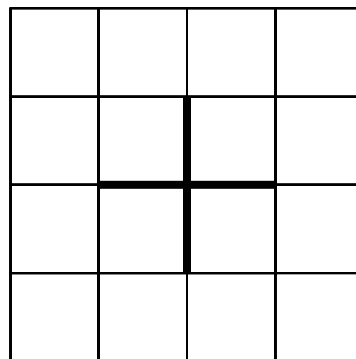
(Real code must account for the special cases rank=0 and rank=npes-1 also.)

EXAMPLE 2: Jacobi Iteration

Laplace Equation: $\nabla^2 T = 0$

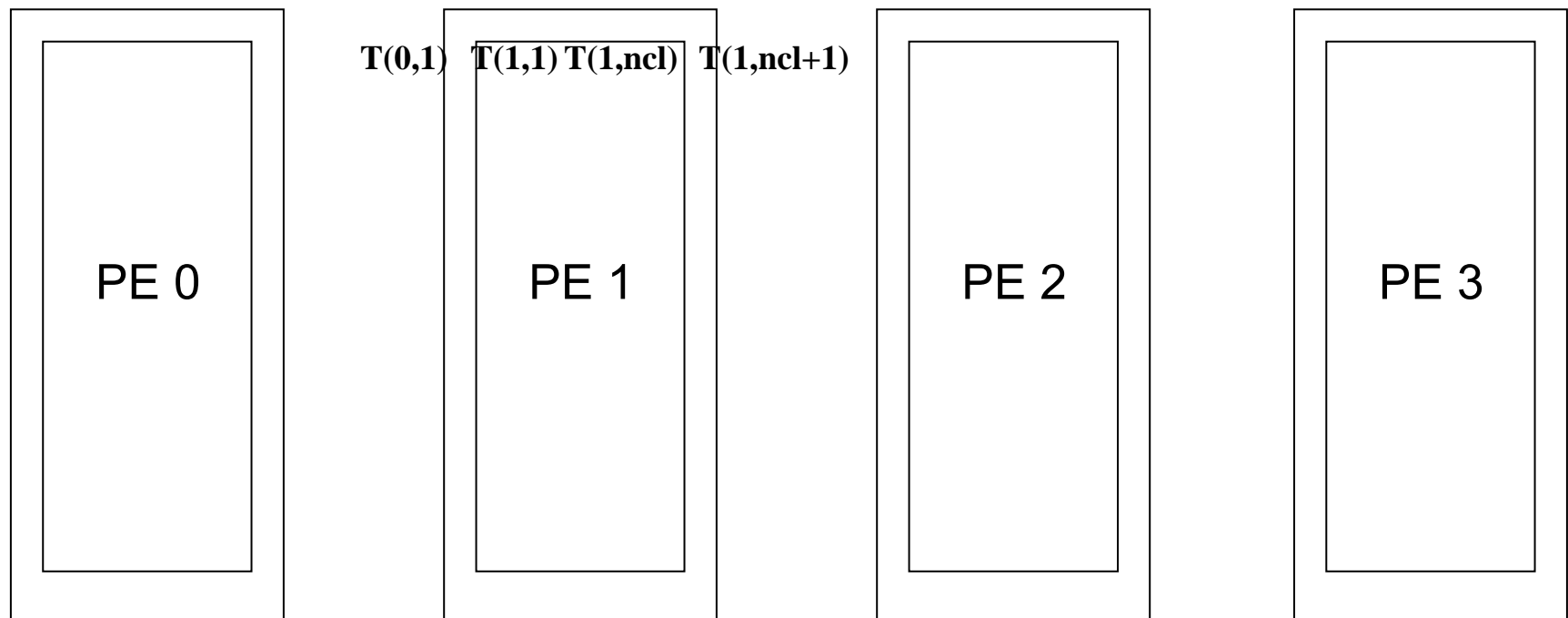
$$T_n = 0.25 * (T_{n-1}(i-1, j) + T_{n-1}(i+1, j) + T_{n-1}(i, j-1) + T_{n-1}(i, j+1))$$

This leads to a five-point stencil



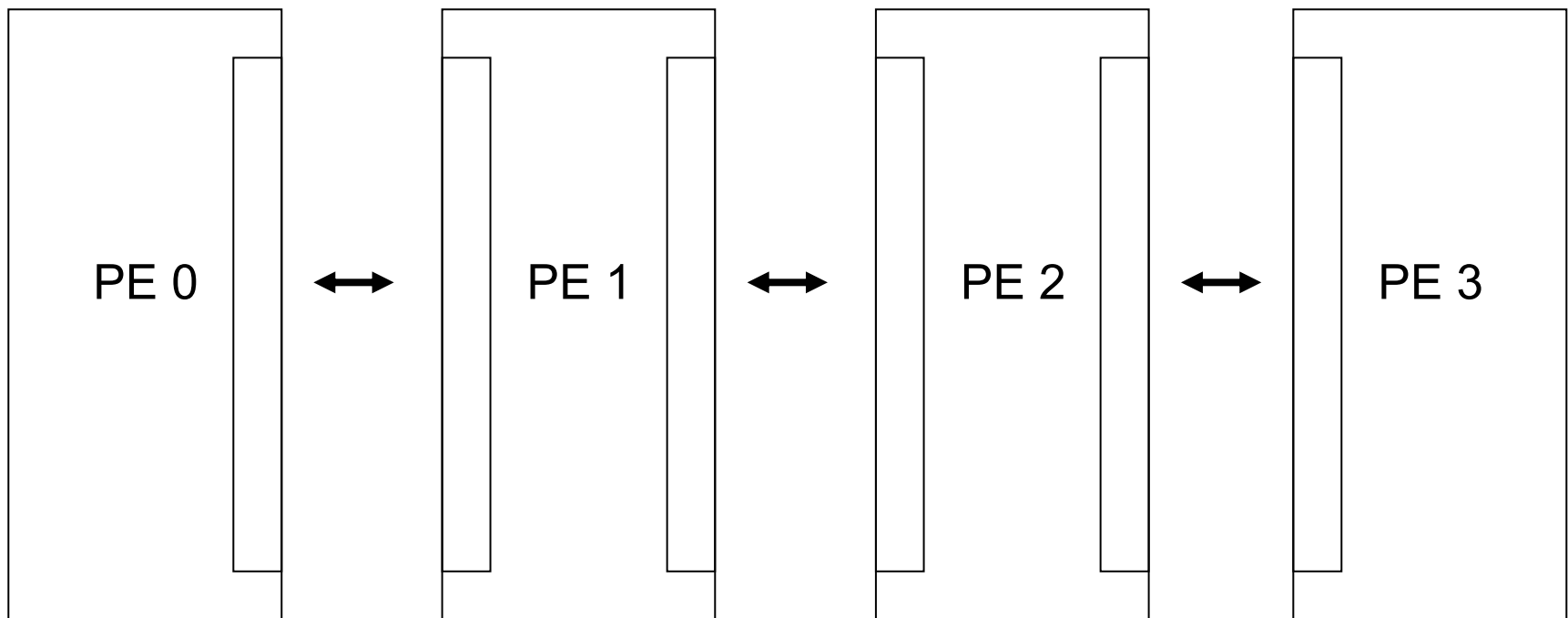
Sample Parallelization Strategy

Break the grid into groups of columns for Fortran, rows for C.



The outer rectangles accommodate boundary and "ghost" zones.

Exchange Edge Data at Each Iteration




```

• else                                ! mype is even
•   if ( mype .gt. 0) then
•     call MPI_RECV(T(1,0) ,nr,MPI_DOUBLE_PRECISION,mype-1,tag, &
•       MPI_COMM_WORLD,status,ierr)
•   endif
•   if ( mype .lt. npes-1) then
•     call MPI_RECV(T(1,ncl),nr,MPI_DOUBLE_PRECISION,mype+1,tag, &
•       MPI_COMM_WORLD,status,ierr)
•   endif
•   if ( mype .gt. 0) then
•     call MPI_SEND(T(1,1) ,nr,MPI_DOUBLE_PRECISION,mype-1,tag, &
•       MPI_COMM_WORLD,status,ierr)
•   endif
•   if ( mype .lt. npes-1) then
•     call MPI_SEND(T(1,ncl),nr,MPI_DOUBLE_PRECISION,mype+1,tag, &
•       MPI_COMM_WORLD,status,ierr)
•   endif
• endif

```

Another Solution

- This pattern of sends and receives is sufficiently common that there is a subroutine
 - `MPI_SENDRECV(sendbuf,count,mpi_type,dest,tag,recvbuf,count,mpi_type,source,tag,comm,status,err)`
- Sends and receives can go to the NULL process `MPI_PROC_NULL`
 - This can simplify code

Nonblocking Sends and Receives

- Beyond the scope of this brief talk
- Allow overlap of computation and communication
- Begin with the character **I** (ISEND, IRECV)
- Completed with WAIT

EXERCISE

- From Aspen, copy `/export/rescomp/mpi_workshop.tar` to your home directory.
- Untar it
 - `tar xf mpi_workshop.tar`
- Cd to trapezoid. Compile and run `trap.f90` (serial version of trapezoid-rule integration code)
 - `module add ifc`
 - `ifc -o trap -cm -w trap.f90`
 - `./trap > trap.out`

Exercise [cont.]

- Compile partrap.f90
 - module add mpich-eth-intel
 - mpif90 -o partrap -cm -w partrap.f90
- Use partrap.sh to submit and run the job. Use npes=1 and npes=2.
- Compare results.

HOMework

- Examine, compile, and run `serjacobi.f90`. This is a Jacobi iteration code.
- Your assignment: take `jacobi.f90` or `jacobi.c` and parallelize it. These codes have indications added for adding MPI calls and what they should do.
- Two possible answers for `jacobi.f90` are in the subdirectory *solution*. No peeking until you've given it a try yourself!

MPI References

- **Parallel Programming with MPI** by Peter Pacheco.
- **MPI: The Complete Reference, Second Edition** by Snir *et al.* PostScript or PDF versions of the First Edition of this manual are available around the Internet, e.g.
 - <http://www.phyast.pitt.edu/beowulf/Tutorial.html>
 - The first edition contains some bugs in example code, but is fine as a reference to the subroutine parameter lists.

Web Resources

- MPI Homepage:
 - <http://www-unix.mcs.anl.gov/mpi>
- Online Tutorial through NCSA:
 - <http://pacont.ncsa.uiuc.edu:8900/public/MPI/>
- Links to PostScript or PDF versions of User's Guides (C and F77/F90)
 - <http://www.phyast.pitt.edu/beowulf/Tutorial.html>
- HTML version of the first edition of the **Complete Reference** :
 - <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>

Parallelization Without MPI

Parallel Libraries

- Well tested and widely used—don't reinvent the wheel.
- If not already present, ITC can install them.
- Examples include
 - ScaLAPACK (linear algebra)
<http://www.netlib.org/scalapack>,
 - PARPACK (eigenvalues/vectors)
http://www.caam.rice.edu/~kristyn/parpack_home.html

ScaLAPACK

- Parallel version of LAPACK Fortran library for linear algebra
- Available on ITC Linux clusters for the Intel compiler
- Reference:
 - http://www.netlib.org/scalapack_home.html

IMSL

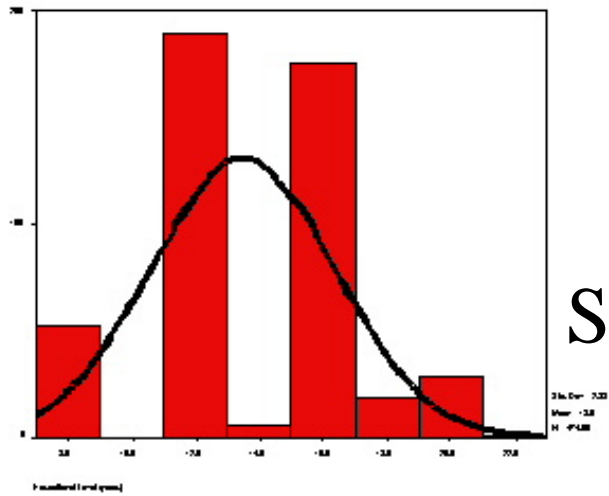
- Commercial library for many algorithms (statistics as well as mathematical)
- Several of the algorithms have parallel versions
- IMSL also provides subroutines to set up the block-cyclic decomposition required by ScaLAPACK
- Use `MPI_SETUP` routine provided by IMSL, not `MPI_INIT`
- Documentation available in
 - `/common/imsl5.0/CTT6.0/help`
 - routines with *gray* background shading can be used in parallel
- `/common/imsl5.0/CTT6.0/examples/linux/f90/mpi_manual`
 - actual code examples on Aspen, Birch, or Cedar

On the Horizon

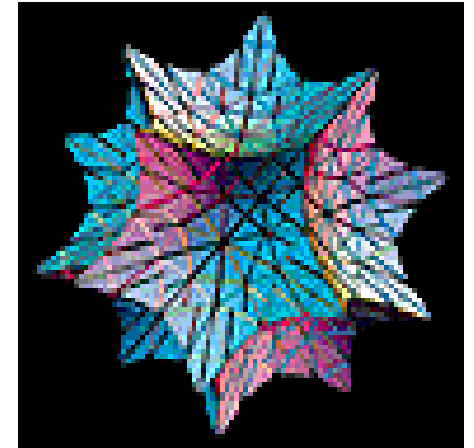
- Co-Array Fortran
- UPC (Unified Parallel C)

These language extensions allow parallel code to be written much more simply, without the need for explicit calls to MPI. However, the compilers are still in an experimental state and are difficult to build and install.

Co-AF to be part of the Fortran 2008 language specification



Some Useful Information



Cedar Hands-on Tutorial is online at www.itc.virginia.edu/research/linux-clusters/cedar/hands-on

Talks are online at www.itc.virginia.edu/research/talks