Introduction to Message Passing Interface

ACF Spring HPC Training Workshop
Match 15-16, 2016
Kwai Wong
Message Passing Interface (MPI)  
Distributive Node level communication

Topics:

- Message Passing Interface
- Point to Point Communication
- Collective Communication
- Derived Datatype
- Parallel Code Development
Message Passing Interface (MPI)

- A first portable message passing communications standard defined by the MPI Forum which consists of hardware vendors, researchers, academics, software developers, and users, representing over forty different organizations

- MPI library (implementation) consists of a set of MPI function calls to facilitate transfer of data across processes

- The syntax of MPI call is standardized

- The functional behavior of MPI call is standardized

- About 20+ calls are in general needed!

- Vendors have tuned MPI calls to suit the underlying hardware (interconnect, arch) and software (OS)

- IBM MPI : IBM implementation for the SP, Cray MPT

- Intel MPI, MPICH, OpenMPI ..
```c
/* Simple serial Hello World C Example : hello.c */
#include <stdio.h>

int main (int argc, char *argv[]) {
    int rank, size;
    printf( "Hello World from process %d of %d\n", rank, size );
    return 0;
}

/* Parallel Hello world C Example : hello.c */
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
    int rank, size;
    MPI_Init (&argc, &argv); /* starts MPI */
    MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* get current process id */
    MPI_Comm_size (MPI_COMM_WORLD, &size); /* get number of processes */
    printf( "Hello World from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

To compile : > icc --o hexe ./hello.c
To compile : > mpiicc --o hexe ./hello.c
To run serial code : > ./hexe
To run in parallel : > mpirun --n 4 ./hexe
SPMD

- Single Program, Multiple Data Programming Paradigm
- Same program runs on all processors, however, each processor operates on different set of data
- The simplest parallel parallel paradigm
- Generally contains:
  
  ```
  if ( my_processor_id .eq. designated_id ) then 
  -------
  end if
  ```
Message Passing Programming

- Used primarily on distributed memory computing environment
- Since memory are local, any data stored in remote processor’s memory must be explicitly requested by programmer.
- Each processor runs the **SAME program** using **partitioned data** set
- Written in sequential language (FORTRAN, C, C++) + MPI functions

```
Sum.exe (P0) :  
  myid =0, N=10
  Data A, B (1:10)
  Do I=1,N
    AL(1:10)=A(myid*N+(1:10))
    BL(1:10)=B(myid*N+(1:10))
    CL=AL+BL
    SUM CL (MPI_Alleduce, C)
  Print global_sum C

Sum.exe (P1) :
  myid =1, N=10
  Data A, B(11:20)
  Do I=1,N
    AL(1:10)=A(myid*N+(1:10))
    BL(1:10)=B(myid*N+(1:10))
    CL=AL+BL
    SUM CL (MPI_Alleduce, C)
  Print global_sum C
```
MPI Message Components

• Envelope:
  - sending processor (processor_id)
  - source location (group_id, tag)
  - receiving processor (processor_id)
  - destination location (group_id, tag)

• Data (letter):
  - data type (integer, float, complex, char....)
  - data length (buffer size, count, strides)
Typical Message Passing Subroutines

- Environment Identifiers
  - processor_id, group_id, initialization
- Point to Point Communications
  - blocking operations
  - non-blocking operations
- Collective Communications
  - barriers
  - broadcast
  - reduction operations
Essentials of MPI programs

- **Header file:**
  - C: `#include<mpi.h>`
  - Fortran: `include 'mpif.h'`

- **Initializing MPI:**
  - C: `int MPI_Init(int argc, char**argv)`
  - Fortran: `call MPI_INIT(IERROR)`

- **Exiting MPI:**
  - C: `int MPI_Finalize()`
  - Fortran: `call MPI_FINALIZE(IERROR)`
MPI Function Format

- **MPI**
  - \texttt{MPI\_Xxxx(\texttt{parameter, \ldots})}

- **C:**
  - \texttt{error = MPI\_Xxxx(\texttt{parameter, \ldots})}
  - Case is IMPORTANT

- **Fortran:**
  - call \texttt{MPI\_XXXX(\texttt{parameter,\ldots, IERROR})}
  - Case is NOT important
MPI Process Identifiers

- **MPI_COMM_RANK:**
  - Gets a process’s rank (ID) within a process group
  - C : int MPI_Comm_rank(MPI_Comm comm, int *rank)
  - F : call MPI_COMM_RANK(mpi_comm,rank,ierror)

- **MPI_COMM_SIZE:**
  - Gets the number of processes within a process group
  - C : int MPI_Comm_size(MPI_Comm comm, int *size)
  - F : call MPI_COMM_SIZE(mpi_comm,size,ierror)

- **MPI_Comm : Communicator**
MPI Communicator

- A communicator is the communicating space among processes
- All MPI communication calls require a communicator argument and MPI processes can only communicate if they share a communicator.
- Every communicator contains a **group of tasks** with a *system supplied identifier* (for message context) as an extra match field.
- MPI_Init initializes all tasks with MPI_COMM_WORLD
- So, the **base group** is the group that contains all processes, which is associated with the MPI_COMM_WORLD communicator.
- Communicators are particularly important for user supplied libraries
- Communicators are used to create independent “message universe”
Communicating Group

ALL : MPI_COMM_WORLD (world)
ROW : comm1, comm2 ; COLUMN : comm3, comm4, comm5
Every process has three communicating groups and a distinct rank associated to it
Sample Program : Hello World

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

main(int argc, char** argv) {
    int my_PE_num;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
    printf("hello from %d. 
", my_PE_num);
    MPI_Finalize();
}
```

program Hello_World

```fortran
program Hello_World

include 'mpif.h'

integer my_PE_num, ierror

call MPI_INIT(ierr)

call MPI_COMM_RANK(MPI_COMM_WORLD, my_PE_num, ierr)

print *, 'Hello from', my_PE_num

call MPI_FINALIZE(ierr)
end
```
Compiling and Running MPI (MPICH PC Cluster)

• Compile:
  – `mpicc -o fileexe filename.c`
  – `mpif77 -o fileexe filename.f`

• Running:
  – `mpirun -np <num_of_processes> fileexe`

• Options:
  – `-np`: number of processes
  – `-machinefile`: list of machines to be used
  – `-t`: testing option
  
  use “man mpirun” to see more details
  %mpirun –np 3 –machinefile hostfile hello.exe
  hello from 0
  hello from 2
  hello from 1
**Compiling and Running – (ACF)**

- Compile (PE-intel): `mpiicc -o hello hello.c` or `mpiifort -o hello hello.f`
- Compile (PE-gnu): `mpicc -o hello hello.c` or `mpif90 -o hello hello.f`
- PBS Batch Script, e.g. `submit.pbs`
- run: `qsub submit.pbs`

```bash
#!/bin/bash
#PBS -A your-project-id
#PBS -N test
#PBS -j oe
#PBS -l walltime=1:00:00,-l nodes=1:ppn=12
cd $PBS_O_WORKDIR
date
mpirun -np 4 ./hello
```
program Hello_World
include ‘mpif.h’
integer me, ierror, ntag, status(MPI_STATUS_SIZE)
character(12) message
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierr)
ntag = 100
if ( me .eq . 0) then
    message = ‘Hello, World’
call MPI_Send(message, 12, MPI_CHARACTER, 1, ntag, 
                MPI_COMM_WORLD, ierr)
else
    call MPI_Recv(message, 12, MPI_CHARACTER, 0, ntag, 
                   MPI_COMM_WORLD, status, ierr)
    print *, ‘node’, me, ‘:’, message
endif
call MPI_FINALIZE(ierr)
end
Parallel Processing

helloworld

P0
- RANK ID
- Message Passing
  - message = Hello World
  - send
  - recv

P1
- RANK ID
- Message Passing
  - message = (empty)
  - recv

B
- Message Passing
  - message = Hello World
  - print

helloworld

P0
- RANK ID
- Message Passing
  - message = Hello World
  - print

P1
- RANK ID
- Message Passing
  - message = Hello World
  - print

Done (exit)
Example: Passing a Message – Hello World Again!

program Hello_World
include 'mpif.h'
integer me, ierror, ntag, status(MPI_STATUS_SIZE)
character(12) message
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierror)
ntag = 100
if ( me .eq . 0) then
  message = 'Hello, World'
call MPI_Send(message, 12, MPI_CHARACTER, 1, ntag, MPI_COMM_WORLD, ierr)
else if ( me .eq . 1) then
  call MPI_Recv(message, 12, MPI_CHARACTER, 0, ntag, MPI_COMM_WORLD, status, ierr)
  print *, 'Node',me, ':', message
endif
call MPI_FINALIZE(ierr)
end
#include "mpi.h"
#include <iostream>
#include <string>

int main(int argc, char ** argv)
{
    int my_PE_num, ntag = 100;
    char message[13] = "Hello, world";
    MPI::Status status;
    MPI::Init(argc, argv);
    my_PE_num = MPI::COMM_WORLD.Get_rank();

    if ( my_PE_num == 0 )
    {
        MPI::COMM_WORLD.Send(message,12,MPI::CHAR,1,ntag);
    }
    else if ( my_PE_num == 1 ) {
        MPI::COMM_WORLD.Recv(message,12,MPI::CHAR,0,ntag, status);
        cout << "Node " << my_PE_num <<" : " << message << endl; }
    MPI::Finalize();
}
MPI Point to Point Communications

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator
- Blocking vs non-blocking calls
- MPI defines four communication modes for blocking and non-blocking send: synchronous, buffered, ready, and standard
- The receive call does not specify communication mode— it is simply blocking and non-blocking.
- Two messages sent from one process to another will arrive in that relative order.
Sending a Message

- **C :: Standard send**
  ```
  int MPI_Send(&buf, count, datatype, dest, tag, comm)
  - &buf : pointer of object to be sent
  - count : the number of items to be sent, e.g. 10
  - datatype : the type of object to be sent, e.g. MPI_INT
  - dest : destination of message (rank of receiver), e.g. 6
  - tag : message tag, e.g. 78
  - comm : communicator, e.g.(MPI_COMM_WORLD)
  ```

- **Fortran ::**
  ```
  call MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
  ```
Receiving a Message

- **C :: Blocking receive**
  
  ```c
  int MPI_Recv(&buf, count, datatype, source, tag, comm, &status)
  ```
  - source: the node to receive from, e.g. 0
  - &status: a structure which contains three fields, the source, tag, and error code of the incoming message.

- **Fortran ::**
  
  ```fortran
  call MPI_RECV(buf, count, datatype, source, tag, comm, status(MPI_STATUS_SIZE), ierror)
  ```
  - status: an array of integers of size MPI_STATUS_SIZE
<table>
<thead>
<tr>
<th>MPI Datatypes</th>
<th>C Datatypes</th>
<th>MPI Datatypes</th>
<th>C Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>---------</td>
<td>MPI_PACKED</td>
<td>---------</td>
</tr>
</tbody>
</table>
# MPI Basic Datatypes - Fortran

<table>
<thead>
<tr>
<th>MPI Datatypes</th>
<th>Fortran Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>---------------</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>---------------</td>
</tr>
</tbody>
</table>
Sample Program : Send and Receive

#include "mpi.h"

main(int argc, char ** argv) {
    int my_PE_num, numtorecv, numtosend=42;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

    if (my_PE_num == 0) {
        MPI_Recv(&numtorecv, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        printf("Number received is : %d\n", numtorecv);
    } else MPI_Send (&numtosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD); 
    MPI_Finalize();
}
**MPI Send**

- **MPI has 8 different types of Send**
- **The nonblocking send has an extra argument of request handle**

<table>
<thead>
<tr>
<th></th>
<th>blocking</th>
<th>nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>synchronous</td>
<td>MPI_Ssend</td>
<td>MPI_ISsend</td>
</tr>
<tr>
<td>buffer</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>ready</td>
<td>MPI_Rsend</td>
<td>MPI_Iresend</td>
</tr>
</tbody>
</table>
Blocking Calls

• A blocking send or receive call suspends execution of user’s program until the message buffer being sent/received is safe to use.

• In case of a blocking send, this means the data to be sent have been copied out of the send buffer, but they have not necessarily been received in the receiving task. The contents of the send buffer can be modified without affecting the message that was sent.

• The blocking receive implies that the data in the receive buffer are valid.
A blocking MPI call means that the program execution will be suspended until the message buffer is safe to use. The MPI standards specify that a blocking SEND or RECV does not return until the send buffer is safe to reuse (for MPI_SEND), or the receive buffer is ready to use (for MPI_RECV).
Non-Blocking Calls

- Non-blocking calls return immediately after initiating the communication.

- In order to reuse the send message buffer, the programmer must check for its status.

- The programmer can choose to block before the message buffer is used or test for the status of the message buffer.

- A blocking or non-blocking send can be paired to a blocking or non-blocking receive
Non-Blocking Send and Receive

- Separate Non-Blocking communication into three phases:
  - Initiate non-blocking communication.
  - Do some work (perhaps involving other communications?)
  - Wait for non-blocking communication to complete.

Non-Blocking Send Diagram:

<table>
<thead>
<tr>
<th>Send Process</th>
<th>Non-Blocking Send Diagram:</th>
<th>Receive Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>MPI</td>
<td>User</td>
</tr>
<tr>
<td>Start send</td>
<td>Complete send (wait)</td>
<td>Receive</td>
</tr>
<tr>
<td>time</td>
<td>message</td>
<td></td>
</tr>
</tbody>
</table>

Non-Blocking Receive Diagram:

<table>
<thead>
<tr>
<th>Send Process</th>
<th>Non-Blocking Receive Diagram:</th>
<th>Receive Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>MPI</td>
<td>User</td>
</tr>
<tr>
<td>Start send</td>
<td>Complete receive (wait)</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>message</td>
<td></td>
</tr>
</tbody>
</table>

User MPI User
User MPI User
User MPI User
User MPI User
User MPI User
Deadlock

- all tasks are waiting for events that haven’t been initiated
- common to SPMD program with blocking communication, e.g. every task sends, but none receives
- insufficient system buffer space is available
- remedies:
  - arrange one task to receive
  - use MPI_Ssendrecv
  - use non-blocking communication
#include <stdio.h>
#include “mpi.h”

main(int argc, char *argv[])
{
    int myrank, nprocs, leftid, rightid, val, sum, tmp;
    MPI_Status recv_status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    if((leftid=(myrank-1)) < 0) leftid = nprocs -1;
    if((rightid=(myrank+1) == nprocs) rightid = 0;
    val = myrank
    sum = 0;
    do {
        MPI_Send(&val,1,MPI_INT,rightid,99, MPI_COMM_WORLD);
        MPI_Recv(&tmp,1, MPI_INT, leftid, 99, MPI_COMM_WORLD, &recv_status);
        val = tmp;
        sum += val;
    } while (val != myrank);
    printf(“proc %d sum = %d \n”, myrank, sum);
    MPI_Finalize();
}
Example: Ring (Blocking Communication) – Schematic

Core 0
Core 1
Core 2
Core 3

Core 0
Core 1
Core 2
Core 3

Core 0
Core 1
Core 2
Core 3

Core 0
Core 1
Core 2
Core 3

myrank=0
size=4
leftid=3
rightid=1
Val=3,2,1,0

myrank=1
size=4
leftid=0
rightid=2
Val=0,3,2,1

myrank=2
size=4
leftid=1
rightid=3
Val=1,0,3,2

myrank=3
size=4
leftid=2
rightid=0
Val=2,1,0,3
Example: Ring (Blocking Communication) – Fortran

```fortran
PROGRAM ring
IMPLICIT NONE
include "mpif.h"
INTEGER ierror, val, my_rank, nprocs, rightid, leftid, tmp, sum, request
INTEGER send_status(MPI_STATUS_SIZE), recv_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, nprocs, ierr)
rightid = my_rank + 1
IF (rightid .EQ. nprocs) rightid = 0
leftid = my_rank - 1
IF (leftid .EQ. -1) leftid = nprocs-1
sum = 0
val = my_rank
100 CONTINUE
CALL MPI_SEND(val, 1, MPI_INTEGER, rightid, 99,
$     MPI_COMM_WORLD, request, ierr)
CALL MPI_RECV(tmp, 1, MPI_INTEGER, leftid, 99,
$     MPI_COMM_WORLD, recv_status, ierr)
sum = sum + tmp
val = tmp
IF(tmp .NE. my_rank) GOTO 100
PRINT *, 'Proc ', my_rank, ' Sum = ',sum
CALL MPI_FINALIZE(ierr)
STOP
END
```
Example: Ring (Non-blocking Communication) – Schematic

myrank=0
size=4
leftid=3
rightid=1

myrank=1
size=4
leftid=0
rightid=2

myrank=2
size=4
leftid=1
rightid=3

myrank=3
size=4
leftid=2
rightid=0
#include <stdio.h>
#include "mpi.h"

main(int argc, char *argv[]) {
    int myrank, nprocs, leftid, rightid, val, sum, tmp;
    MPI_Status recv_status, send_status;
    MPI_request send_request;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    if((leftid=(myrank-1)) < 0) leftid = nprocs - 1;
    if((rightid=(myrank+1) == nprocs) rightid = 0;
    val = myrank
    sum = 0;
    do {
        MPI_Issend(&val,1,MPI_INT,right,99, MPI_COMM_WORLD,&send_request);
        MPI_Recv(&tmp,1, MPI_INT, left, 99, MPI_COMM_WORLD, &recv_status);
        MPI_Wait(&send_request,&send_status);
        val = tmp;
        sum += val;
    } while (val != myrank);
    printf("proc %d sum = %d \n", myrank, sum);
    MPI_Finalize();
}
Example: Ring (Non-blocking Communication) – Fortran

```
PROGRAM ring
IMPLICIT NONE
include "mpif.h"
INTEGER ierror, val, my_rank, nprocs, rightid, leftid, tmp, sum
INTEGER send_status(MPI_STATUS_SIZE), recv_status(MPI_STATUS_SIZE)
INTEGER request
CALL MPI_INIT(ierror)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierror)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierror)
rightid = my_rank + 1
IF (rightid .EQ. nprocs) rightid = 0
leftid = my_rank - 1
IF (leftid .EQ. -1) leftid = nprocs - 1
sum = 0
val = my_rank
100 CONTINUE
CALL MPI_ISSEND(val, 1, MPI_INTEGER, rightid, 99,
$     MPI_COMM_WORLD, request, ierror)
CALL MPI_RECV(tmp, 1, MPI_INTEGER, leftid, 99,
$     MPI_COMM_WORLD, recv_status, ierror)
CALL MPI_WAIT(request, send_status, ierror)
sum = sum + tmp
val = tmp
IF (tmp .NE. my_rank) GOTO 100
PRINT *, 'Proc ', my_rank, ' Sum = ', sum
CALL MPIFINALIZE(ierr)
STOP
END
```
Example: MPI Communication Timing Test

- The objective of this exercise is to investigate the amount of time required for message passing between two processes, i.e. an MPI communication timing test is performed.
- In this exercise different size messages are sent back and forth between two processes a number of times. Timings are made for each message before it is sent and after it has been received. The difference is computed to obtain the actual communication time. Finally, the average communication time and the bandwidth are calculated and output to the screen.
- For example, one can run this code on two nodes (one process on each node) passing messages of length 1, 100, 10,000, and 1,000,000 and record the results in a table.
Sendrecv

- useful for executing a shift operation across a chain of processes
- system take care of possible deadlock due to blocking calls

`MPI_Sendrecv(sbuf, scount, stype, dest, stag, rbuf, rcount, rtype, source, rtag, comm, status)`

- `sbuf (rbuf)`: initial address of send (receive) buffer
- `scount (rcount)`: number of elements in send (receive) buffer
- `stype (rtype)`: type of elements in send (receive) buffer
- `stag (rtag)`: send (receive) tag
- `dest`: rank of destination
- `source`: rank of source
- `comm`: communicator
- `status`: status object
Non-blocking Communications

- The non-blocking calls have the same syntax as the blocking calls, with two exceptions:
  - Each call has an “I” immediately following the “_”
  - The last argument is a handle to an opaque request object that contains information about the message

- Non-blocking call returns immediately after initiating the communication

- The programmer can block or check for the status of the message buffer: MPI_Wait or MPI_Test.
Non-blocking Send and Receive

- **Fortran**:
  ```fortran
call MPI_Isend(buf,count,datatype,dest,tag,comm,handle,ierr)
call MPI_Irecv(buff,count,datatype,src,tag,comm,handle,ierr)
call MPI_Test(handle, flag, status, ierr)
call MPI_Wait (handle, status, ierr)
```

- **C**:
  ```c
  MPI_Isend(&buf, count, datatype, dest, tag, comm, &handle)
  MPI_Irecv(&buff, count, datatype, src, tag, comm, &handle)
  MPI_Wait (&handle, &status)
  MPI_Test(&handle, &flag, &status)
  ```
MPI_Wait (request, status)

These routines block until the communication has completed. They are useful when the data from the communication buffer is about to be re-used.

MPI_Test (request, flag, status)

This routine blocks until the communication specified by the handle request has completed. The request handle will have been returned by an earlier call to a non-blocking communication routine. The routine queries completion of the communication and the result (TRUE or FALSE) is returned in flag.
Timer

- C
  
  double MPI_Wtime(void)

- Fortran:
  
  double precision MPI_Wtime()

- Time is measured in seconds.

- Time to perform a task is measured by consulting the timer before and after

- Modify your program to measure its execution time and print it out
Inner Product (Parallel)

- Compute the inner product of two vectors,
  \[ sdot = \sum_{i=1}^{n} x^T y \]

- Blockwise distribution of the vectors is used. Each processor computes a portion of the value of the inner product. The result is obtained by summing the partial values together.
Matrix - Vector Product

- Compute the product of a matrix, A, and a vector, x, \( y = y + Ax \)

- Matrix A can be distributed to processors with a block-striped partition scheme or a block cyclic scheme. Entire vector x will be needed for calculation of y in every processor. Vector x may be distributed or duplicated among processors. Each processor computes the value of a portion of the matrix vector product. The resultant y vector is obtained by summing the partial values.
Parallel M-V Multiplication

Processor 0

\[
\begin{bmatrix}
\text{[filled]} & \text{[filled]} & \text{[filled]} \\
\text{[filled]} & \text{[filled]} & \text{[filled]} \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\text{[question]} \\
\text{[question]} \\
\end{bmatrix}
\]

= 

\[
\begin{bmatrix}
\text{[question]} \\
\end{bmatrix}
\]

Processor 1

\[
\begin{bmatrix}
\text{[filled]} & \text{[filled]} & \text{[filled]} \\
\text{[filled]} & \text{[filled]} & \text{[filled]} \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\text{[question]} \\
\text{[question]} \\
\end{bmatrix}
\]

= 

\[
\begin{bmatrix}
\text{[question]} \\
\end{bmatrix}
\]
Parallel M-V Multiplication
Data Mapping Scheme

- The matrix A can be distributed to processors in a checkerboard partition, a two dimensional block-block distribution.

\[
\begin{array}{ccc}
(0,0) & (0,1) & (0,2) \\
(1,0) & (1,1) & (1,2) \\
(2,0) & (2,1) & (2,2) \\
\end{array} \times \begin{array}{c}
(*,0) \\
(*,1) \\
(*,2) \\
\end{array} + \begin{array}{c}
(0,*) \\
(1,*) \\
(2,*) \\
\end{array} = \begin{array}{c}
(0,*) \\
(1,*) \\
(2,*) \\
\end{array}
\]

- For an nxn matrix, n^2 processors can be used. The vector x is distributed to the processors sharing the same column number. The partial products computed for each row are added together to obtain the resultant vector y.
- This mapping scheme scales more efficiently because a larger number of processors can be used, however, it requires additional amount of communication work.
**Simple Matrix Multiplication Algorithm**

- Matrix A is copied to every processors (FORTRAN)
- Matrix B is divided into blocks and distributed to processors
- Perform matrix multiplication simultaneously
- Output solutions

\[ \begin{array}{cccc}
\text{proc} & 1, 2, 3, 4 \\
\hline
\text{p1} & \text{p2} & \text{p3} & \text{p4} \\
\hline
A & \times & B & = C
\end{array} \]
Parallel Processing

Obtain
A(1:8, 1-8)
B(1:8, 1:8)
A(1:8, 1-8)

B(1:8, 7-8)
B(1:8, 5-6)
B(1:8, 3-4)
B(1:8, 1-2)

C(1:8, 7-8)
C(1:8, 5-6)
C(1:8, 3-4)
C(1:8, 1-2)

C = A * B
C = A * B
C = A * B
C = A * B
Matrix Multiplication

step 1

step 2
Matrix Multiplication (continued)

**Step 3**

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
</tr>
</thead>
<tbody>
<tr>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Step 4**

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fox’s Algorithm (1)

- Broadcast the diagonal element of block A in rows, perform multiplication.

\[
\begin{align*}
A(0,0) & \times B(0,0) = C(0,0) \\
A(0,0) & \times B(0,1) = C(0,1) \\
A(0,0) & \times B(0,2) = C(0,2) \\
A(1,1) & \times B(1,0) = C(1,0) \\
A(1,1) & \times B(1,1) = C(1,1) \\
A(1,1) & \times B(1,2) = C(1,2) \\
A(2,2) & \times B(2,0) = C(2,0) \\
A(2,2) & \times B(2,1) = C(2,1) \\
A(2,2) & \times B(2,2) = C(2,2)
\end{align*}
\]

- \[C(0,0) = A(0,0)B(0,0) + A(0,1)B(1,0) + A(0,2)B(2,0)\]
- \[C(0,1) = A(0,0)B(0,1) + A(0,1)B(1,1) + A(0,2)B(2,1)\]
- \[C(0,2) = A(0,0)B(0,2) + A(0,1)B(1,2) + A(0,2)B(2,2)\]
- \[C(1,0) = A(1,0)B(0,0) + A(1,1)B(1,0) + A(1,2)B(2,0)\]
- \[C(1,1) = A(1,0)B(0,1) + A(1,1)B(1,1) + A(1,2)B(2,1)\]
- \[C(1,2) = A(1,0)B(0,2) + A(1,1)B(1,2) + A(1,2)B(2,2)\]
- \[C(2,0) = A(2,0)B(0,0) + A(2,1)B(1,0) + A(2,2)B(2,0)\]
- \[C(2,1) = A(2,0)B(0,1) + A(2,1)B(1,1) + A(2,2)B(2,1)\]
- \[C(2,2) = A(2,0)B(0,2) + A(2,1)B(1,2) + A(2,2)B(2,2)\]
**Fox’s Algorithm (2)**

- Broadcast next element of block A in rows, shift Bij in column, perform multiplication.

<table>
<thead>
<tr>
<th></th>
<th>A(0,1)</th>
<th>A(0,1)</th>
<th>A(0,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1,2)</td>
<td>A(1,2)</td>
<td>A(1,2)</td>
<td></td>
</tr>
<tr>
<td>A(2,0)</td>
<td>A(2,0)</td>
<td>A(2,0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B(1,0)</th>
<th>B(1,1)</th>
<th>B(1,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2,0)</td>
<td>B(2,1)</td>
<td>B(2,2)</td>
<td></td>
</tr>
<tr>
<td>B(0,0)</td>
<td>B(0,1)</td>
<td>B(0,2)</td>
<td></td>
</tr>
</tbody>
</table>

\[ X = \begin{align*}
C(0,0) &= A(0,0)B(0,0) + A(0,1)B(1,0) + A(0,2)B(2,0) \\
C(0,1) &= A(0,0)B(0,1) + A(0,1)B(1,1) + A(0,2)B(2,1) \\
C(0,2) &= A(0,0)B(0,2) + A(0,1)B(1,2) + A(0,2)B(2,2) \\
C(1,0) &= A(1,0)B(0,0) + A(1,1)B(1,0) + A(1,2)B(2,0) \\
C(1,1) &= A(1,0)B(0,1) + A(1,1)B(1,1) + A(1,2)B(2,1) \\
C(1,2) &= A(1,0)B(0,2) + A(1,1)B(1,2) + A(1,2)B(2,2) \\
C(2,0) &= A(2,0)B(0,0) + A(2,1)B(1,0) + A(2,2)B(2,0) \\
C(2,1) &= A(2,0)B(0,1) + A(2,1)B(1,1) + A(2,2)B(2,1) \\
C(2,2) &= A(2,0)B(0,2) + A(2,1)B(1,2) + A(2,2)B(2,2)
\end{align*} \]
Fox’s Algorithm (3)

• Broadcast next element of block A in rows, shift Bij in column, perform multiplication.

\[
\begin{array}{ccc}
A(0,2) & A(0,2) & A(0,2) \\
A(1,0) & A(1,0) & A(1,0) \\
A(2,1) & A(2,1) & A(2,1) \\
\end{array} \times \begin{array}{ccc}
B(2,0) & B(2,1) & B(2,2) \\
B(0,0) & B(0,1) & B(0,2) \\
B(1,0) & B(1,1) & B(1,2) \\
\end{array} = \begin{array}{ccc}
C(0,0) \\
C(0,1) \\
C(0,2) \\
\end{array} \\
\begin{array}{ccc}
C(1,0) \\
C(1,1) \\
C(1,2) \\
\end{array} \\
\begin{array}{ccc}
C(2,0) \\
C(2,1) \\
C(2,2) \\
\end{array}
\]

• \( C(0,0) = A(0,0)B(0,0) + A(0,1)B(1,0) + A(0,2)B(2,0) \)
• \( C(0,1) = A(0,0)B(0,1) + A(0,1)B(1,1) + A(0,2)B(2,1) \)
• \( C(0,2) = A(0,0)B(0,2) + A(0,1)B(1,2) + A(0,2)B(2,2) \)
• \( C(1,0) = A(1,0)B(0,0) + A(1,1)B(1,0) + A(1,2)B(2,0) \)
• \( C(1,1) = A(1,0)B(0,1) + A(1,1)B(1,1) + A(1,2)B(2,1) \)
• \( C(1,2) = A(1,0)B(0,2) + A(1,1)B(1,2) + A(1,2)B(2,2) \)
• \( C(2,0) = A(2,0)B(0,0) + A(2,1)B(1,0) + A(2,2)B(2,0) \)
• \( C(2,1) = A(2,0)B(0,1) + A(2,1)B(1,1) + A(2,2)B(2,1) \)
• \( C(2,2) = A(2,0)B(0,2) + A(2,1)B(1,2) + A(2,2)B(2,2) \)
Collective Communications

- Substitutes for a more complex sequence of p-p calls
- Involve all the processes in a process group
- Called by all processes in a communicator
- All routines block until they are locally complete
- Receive buffers must be exactly the right size
- No message tags are needed
- Divided into three subsets:
  - synchronization, data movement, and global computation
Barrier Synchronization Routines

- To synchronize all processes within a communicator
- A node calling it will be blocked until all nodes within the group have called it.

- C:
  ```c
  ierr = MPI_Barrier(comm)
  ```

- Fortran:
  ```fortran
call MPI_Barrier(comm, ierr)
  ```
Broadcast

- one processor sends some data to all processors in a group

- C
  ```c
  ierr = MPI_Bcast(buffer, count, datatype, root, comm)
  ```

- Fortran
  ```fortran
call MPI_Bcast(buffer,count,datatype, root, comm, ierr)
  ```

- The MPI_Bcast must be called by each node in a group, specifying the same communicator and root. The message is sent from the root process to all processes in the group, including the root process.
Scatter

- Data are distributed into n equal segments, where the ith segment is sent to the ith process in the group which has n processes.

- C:
  
  ```c
  ierr = MPI_Scatter(&sbuff, scount, sdatatype, &rbuf, rcount, rdatatype, root, comm)
  ```

- Fortran:
  
  ```fortran
  call MPI_Scatter(sbuff, scount, sdatatype, rbuf, rcount, rdatatype, root, comm, ierr)
  ```
Example: Scatter

real sbuf(12), rbuf(2)

call MPI_Scatter(sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 3, MPI_COMM_WORLD, ierr)
Scatter and Gather

DATA

<table>
<thead>
<tr>
<th>PE 0</th>
<th>PE 1</th>
<th>PE 2</th>
<th>PE 3</th>
<th>PE 4</th>
<th>PE 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
</tr>
</tbody>
</table>

scatter

gather

DATA

<table>
<thead>
<tr>
<th>PE 0</th>
<th>PE 1</th>
<th>PE 2</th>
<th>PE 3</th>
<th>PE 4</th>
<th>PE 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>A5</td>
</tr>
</tbody>
</table>


• Data are collected into a specified process in the order of process rank, reverse process of scatter.

• C :
  
  ```c
  ierr = MPI_Gather(&sbuf, scount, sdatatype, &rbuf, rcount, rdatatype, root, comm)
  ```

• Fortran :
  
  ```fortran
  call MPI_Gather(sbuff, scount, sdatatype, rbuff, rcount, rdatatype, root, comm, ierr)
  ```
Example: Gather

real sbuf(2), rbuf(12)

call MPI_Gather(sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 3, MPI_COMM_WORLD, ierr)
Scatterv and Gatherv

- allow varying count of data and flexibility for data placement

- C:

  ```c
  ierr = MPI_Scatterv( &sbuf, &scount, &displace, sdatatype, 
                        &rbuf, rcount, rdatatype, root, comm)
  ierr = MPI_Gatherv(&sbuf, scount, sdatatype, &rbuf, &rcount, 
                      &displace, rdatatype, root, comm)
  ```

- Fortran:

  ```fortran
  call MPI_Scatterv(sbuf,scount,displace,sdatatype, rbuf, rcount, 
                   rdatatype, root, comm, ierr)
  ```
Allgather

```c
ierr = MPI_Allgather(&sbuf, scount, stype,&rbuf, rcount, rtype, comm)
```

| PE 0 | A0 |   |   |   |   | DATA |   | A0 | B0 | C0 | b0 | E0 | F0 | PE 0 |
| PE 1 | B0 |   |   |   |   | DATA |   | A0 | B0 | C0 | b0 | E0 | F0 | PE 1 |
| PE 2 | C0 |   |   |   |   | DATA |   | A0 | B0 | C0 | b0 | E0 | F0 | PE 2 |
| PE 3 | b0 |   |   |   |   | DATA |   | A0 | B0 | C0 | b0 | E0 | F0 | PE 3 |
| PE 4 | E0 |   |   |   |   | DATA |   | A0 | B0 | C0 | b0 | E0 | F0 | PE 4 |
| PE 5 | F0 |   |   |   |   | DATA |   | A0 | B0 | C0 | b0 | E0 | F0 | PE 5 |
All to All

MPI_Alltoall(sbuf, scount, stype, rbuf, rcount, rtype, comm)

- **sbuf**: starting address of send buffer (*)
- **scount**: number of elements sent to each process
- **stype**: data type of send buffer
- **rbuff**: address of receive buffer (*)
- **rcount**: number of elements received from any process
- **rtype**: data type of receive buffer elements
- **comm**: communicator
All to All

PE 0  PE 1  PE 2  PE 3  PE 4  PE 5

DATA
A0  A1  A2  A3  A4  A5
B0  B1  B2  B3  B4  B5
C0  C1  C2  C3  C4  C5
D0  D1  D2  D3  D4  D5
E0  E1  E2  E3  E4  E5
F0  F1  F2  F3  F4  F5

alltoall

PE 0  PE 1  PE 2  PE 3  PE 4  PE 5

DATA
A0  A1  A2  A3  A4  A5
B0  B1  C0  D0  E0  F0
A1  B1  C1  D1  E1  F1
A2  B2  C2  D2  E2  F2
A3  B3  C3  D3  E3  F3
A4  B4  C4  D4  E4  F4
A5  B5  C5  D5  E5  F5
Global Reduction Routines

• The partial result in each process in the group is combined together using some desired function.

• The operation function passed to a global computation routine is either a predefined MPI function or a user supplied function.

• Examples:
  – global sum or product
  – global maximum or minimum
  – global user-defined operation
Reduce and Allreduce

- **MPI_Reduce(sbuf, rbuf, count, stype, op, root, comm)**
  - **sbuf**: address of send buffer
  - **rbuf**: address of receive buffer
  - **count**: the number of elements in the send buffer
  - **stype**: the datatype of elements of send buffer
  - **op**: the reduce operation function, predefined or user-defined
  - **root**: the rank of the root process
  - **comm**: communicator

- **MPI_Allreduce(sbuf, rbuf, count, stype, op, comm)**
  - **sbuf**: address of send buffer
  - **rbuf**: address of receive buffer
  - **count**: the number of elements in the send buffer
  - **stype**: the datatype of elements of send buffer
  - **op**: the reduce operation function, predefined or user-defined
  - **root**: the rank of the root process
  - **comm**: communicator

-- mpi_reduce returns results to single process
-- mpi_allreduce returns results to all processes in the group
### Predefined Reduce Operations

<table>
<thead>
<tr>
<th>MPI NAME</th>
<th>FUNCTION</th>
<th>MPI NAME</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>MPI_MAXLOC</td>
<td>Maximum and location</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Bitwise AND</td>
<td>MPI_MINLOC</td>
<td>Minimum and location</td>
</tr>
</tbody>
</table>
Example of Reduce w/Sum

A routines that computes the dot product of two vectors that are distributed across a group of processes and returns the answer to node zero

```fortran
subroutine PAR_BLAS1( m , a , b , c , comm)
    real a(m) , b(m) , sum , c
    integer m , comm , l , ierr

    sum = 0.0
    do l = 1 , m
        sum = sum + a(l) * b(l)
    end do

    call MPI_Reduce( sum , c , 1 , MPI_REAL , MPI_SUM,0, comm , ierr )
    return
```

Derived Datatypes

- To provide a portable and efficient way of communicating non-contiguous or mixed types in a message
- datatypes that are built from the basic MPI datatypes.
- MPI datatypes are created at run-time through calls to MPI library

Steps required
  - construct the datatype : define shapes and handle
  - allocate the datatype : commit types
  - use the datatype : use constructors
  - deallocate the datatype : free types
Datatypes

• Basic datatypes:
  – MPI_INT, MPI_REAL, MPI_DOUBLE, MPI_COMPLEX,
  – MPI_LOGICAL, MPI_CHARACTER, MPI_BYTE ....

• MPI also supports array sections and structures through general datatypes. A general datatypes is a sequence of basic datatypes and integer byte displacements. These displacements are taken to be relative to the buffer that the basic datatype is describing. ==> typemap
  – Datatype = { (type0, disp0) , (type1, disp1) , ......, (typeN, dispN)}
Defining Datatypes

MPI_Type_contiguous(count, oldtype, newtype, ierr)
- ‘count’ copies of ‘oldtype’ are concatenated

MPI_Type_vector(count, buffer, strides, oldtype, newtype, ierr)
- ‘count’ blocks with ‘blen’ elements of ‘oldtype’ spaced by ‘stride’

MPI_Type_indexed(count, buffer, strides, oldtype, newtype, ierr)
- Extension of vector: varying ‘blens’ and ‘strides’

MPI_Type_struct(count, buffer, strides, oldtype, newtype, ierr)
- Extension of indexed: varying data types allowed
Parallel Code Development
Example: Laplace Equation

\[ \nabla^2 T = \frac{\partial^2 T}{\partial X^2} + \frac{\partial^2 T}{\partial Y^2} = 0 \]

5-POINT FD STENCIL

POINT JACOBI ITERATION

\[ T(I, J) = 0.25 \times \{ T(\text{old}(I-1, J)) \]
\[ + T(\text{old}(I+1, J)) \]
\[ + T(\text{old}(I, J-1)) \]
\[ + T(\text{old}(I, J+1)) \} \]
Initial guess : $T = 0$, 
Boundaries : $T = 100$, 
Grid : 1000x1000

Fortran:

```fortran
Do J = 1 , NC
    Do I = 1 , NR
        T ( I , J ) = 0.25 * (Told ( I -1 ,J ) + Told ( I + 1 ,J) + Told( I , J-1 ) + Told(I, J+1) )
    end do
end do
```

C :

```c
for ( I=1 ; I <=NR ; I++)
    for (j=1 ; j <= NC ; j++ )
        T[I][j] = 0.25*( Told[I+1][j] + Told[I-1][j] + Told[I][j+1] + Told[I][j-1] );
```
Domain Decomposition

The diagram illustrates a domain decomposition with four processing elements (PEs) labeled as PE 0, PE 1, PE 2, and PE 3. The domain is divided into subdomains, each assigned to a PE. The coordinates (0,0), (0,1001), (1000,1), and (1000,1000) define the boundaries of the domain. The subdomains are further divided into smaller grid structures, with each PE handling a portion of the domain.
Program formulation

- Include file
- Initialization
- Processor information
- Iteration: {
  - do averaging
  - copy T into Told
  - send values down
  - send values up
  - receive values from above
  - receive values from below
  - find the max change
  - synchronize }
Processor Information

• **Header:**
  #include <mpi.h>  or include ‘mpif.h’

• **Initialization :**
  call MPI_Init(ierr)
  ierr = MPI_Init( & argc, & argv)

• **Number of processors:**
  call MPI_Comm_size( MPI_COMM_WORLD, npes, ierr )
  ierr = MPI_Comm_size( MPI_COMM_WORLD, &npes)

• **Processor number :**
  call MPI_Comm_rank(MPI_COMM_WORLD, mype, ierr)
  ierr= MPI_Comm_rank(MPI_COMM_WORLD, &mype)
Initialization

Serial (interior)

for (I = 0 ; I <= NR + 1 ; I++)
  for ( J = 0 ; <= NC +1 ; J++)
    T[I][J] = 0.0;

do I = 0 , NR +1
  do J =0 , NC + 1
    T( I, J ) = 0.0
  enddo
enddo

Parallel (boundaries)

/* Left and right boundaries */
for (I = 0; I <= NRL + 1; I++) {
  T[I][0] = 100.0;
  T[I][NCL+1] =100.0; }

/*Top and Bottom Boundaries*/
if (MYPE == 0)
  for (J = 0; J <=NCL+1; j++)
    T[0][J] = 100.0;
if (MYPE == NPES - 1)
  for (J = 0 ; J <= NCL +1 ; J++)
    T[NRL+1][J] = 100.0
Do Averaging

NRL : NR / NPES

Fortran :
   Do J = 1 , NC
      Do I = 1 , NRL
         T ( I , J ) = 0.25 * (Told ( I -1 ,J ) + Told ( I + 1 ,J) + Told( I , J-1 )
                     + Told(I, J+1) )
      end do
   end do

C :
   for ( I=1 ;  I <=NRL ;  I++)
      for (j=1 ; j <= NC ; j++)
         T[I][j] = 0.25*( Told[I+1][j] + Told[I-1][j] + Told[I][j+1] +
                         Told[I][j-1] );
Parallel Send

send first row to above

send last row to below
Updating T Values at Domain Boundaries (Send)

Sending up:

```
If (mype != 0) MPI_Send( &t[1][1], NC, MPI_FLOAT, mype-1, UP, MPI_COMM_WORLD )
```

Sending down:

```
If (mype < npes-1) MPI_Send( &t[NRL][1], NC, MPI_FLOAT, mype+1, UP, MPI_COMM_WORLD)
```
Parallel Template: Receive

PE 0

PE 1

PE 2

PE 3

receive first row from above

PE 0

PE 1

PE 2

PE 3

receive last row from below
Updating Values at Domain Boundaries (Receive)

Receiving from up:

```c
If (mype != 0)    MPI_Recv( &t[0][1], NC, MPI_FLOAT,
                           MPI_ANY_SOURCE, DOWN, &status, MPI_COMM_WORLD )
```

Receiving from down:

```c
If (mype != npes-1)    MPI_Recv( &t[NRL+1][1], NC, MPI_FLOAT,
                                MPI_ANY_SOURCE, UP, &status, MPI_COMM_WORLD)
```
Find Maximum Change

- each processor finds its own maximum change, $dt$
- find the global change using reduce, $dt_g$

```c
MPI_Reduce (&dt, &dtg, 1, MPI_FLOAT, MPI_MAX, PE0, comm)
call MPI_Reduce( dt, dtg, 1, MPI_FLOAT, MPI_MAX, PE0, comm, ierr)
```

- synchronization :
  ```c
  MPI_Barrier(MPI_COMM_WORLD)
call MPI_Barrier(MPI_COMM_WORLD, ierr)
  ```
Data Distribution

Columnwise domain decomposition with grid overlap (FORTRAN)  X-Y 2D domain decomposition with grid overlap
Performance Evaluation of Codes

• Profiling of codes
  – Use profiling tools, e.g. prof, to identify segment of code that requires intensively computing
  – craypat on the XT5

• Timing codes
  – timers depend on machines platform
  – $ time executable ---> cpu time, elapsed time
  – etime, dtime, ctime, itime
  – should provide good timing resolution for section of codes

• MPI timer
  – MPI_Wtime() ---> return seconds of elapse wall-clock time
  – MPI_Wtick() ----> return value of seconds between successive clock ticks
Communication Issues

- **Contentions, or traffic jams**
  - Have good distribution of messages. Circular or round-robin methods in one or two dimensions are fairly efficient for certain problems.
  - Avoid as much as possible the use of indirect addressing.
  - Use threads on multicore

- **Ready mode in MPI or post receive before send**
  - use MPI_Rsend when you are *sure* that a matching receive (MPI_Recv) has been posted appropriately
  - this allows faster transfer protocols
  - HOWEVER! behavior is undefined if receive was not posted in time!
  - Post receive before send on Cray

- **Mask communication with computation**
  - Use asynchronous mode,
  - Avoid barrier
I/O and Parallel I/O

- I/O can be a serious bottleneck for certain applications. The time to read/write data to disks could be an issue. But sometimes the sheer size of the data file is a problem.

- Parallel I/O systems allow (in theory) the efficient manipulation of huge files.

- Unfortunately, parallel I/O is only available on some architectures, and software is not always good. (MPI-2 has parallel MPI-IO on ROMIO implementation)

- They are restricted to few (around 4 or so) parallel disk drives, through designated I/O nodes.

- On the IBM with GPFS

- Lustre on the CRAY XT5

- One single files vs file/process

- Using local /tmp for input output

- Progress is still needed in this area!
Mapping Problem

- Each processor should have a similar amount of work
- Expensive communications should be minimized.
- Communications should be:
  - eliminated where feasible
  - localized otherwise (i.e. communicate between close CPU neighbors) 
    (not crucial anymore)
- Concurrency should be maximized
- **NOTE:** finding the best mapping is an NP-complete problem! :-(
The End

Quote: “I think there is a world market for maybe five computers”
   Thomas Watson, chairman of IBM, 1943