Intermediate MPI: A Practical Approach for Programmers (Day 4)

Dr. David Ennis
Ohio Supercomputer Center
and
Dr. Kadin Tseng
Boston University

http://webct.ncsa.uiuc.edu:8900/

Table of Contents

Fortran 90 and MPI-1

- Introduction
- Fortran 90 Syntax
- Fortran 90 Records
- KIND Variables
- Allocatable Arrays
- Arrays Sections
- Array Indexing
- Pointer and Targets

Introduction

- Socratic chapter approach
- In each section a question is asked, then answer provided, and finally sample code
- "I am writing a code using this Fortran 90 feature, how can I run it in parallel with MPI-1?"

New Fortran 90 Syntax

- Q: "Can I even use MPI routines in a Fortran 90 program?"
  - The most basic question of all
- A: Yes, because Fortran 90 is backward-compatible with Fortran 77
- In the sample program, two MPI processes exchange arrays, but all the code syntax is Fortran 90

Sample Program

```
integer :: count
real    :: data(10),value(20)
integer :: err,rank,count,status(MPI_STATUS_SIZE)
call MPI_INIT(err)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
if (rank==1) then
  data=3.0
  call MPI_SEND(data,10,MPI_REAL,0,55,MPI_COMM_WORLD,err)
else
  call MPI_RECV(value,20,MPI_REAL,MPI_ANY_SOURCE,55, &
                MPI_COMM_WORLD, status,err)
  print *,"P:",rank," Message from proc ",status(MPI_SOURCE)
  call MPI_GET_COUNT(status,MPI_REAL,count,err)
  print *,"P:",rank," got ",count," elements"
  print *,"P:",rank," value array is"
  print *,value
end if
```

Program Output

```
P:0  Message from proc 1
P:0  got 10 elements
P:0  value array is
  3.000000        3.000000        3.000000
  3.000000        3.000000
  3.000000        3.000000        3.000000
  3.000000        3.000000
  0.000000E+00    0.000000E+00    0.000000E+00
  0.000000E+00    0.000000E+00
  0.000000E+00    0.000000E+00    0.000000E+00
  0.000000E+00    0.000000E+00
```

Fortran 90 Records

- Q: “Can I transfer Fortran 90 records variables with MPI routines?”
- A: Yes, because a user can define their own MPI_datatype that matches the memory stencil of the record
- Sample program will work with a record type *card* which has two members representing the number and suit of a playing card

Sample Program (Creating MPI_Card)

```
type card
  integer :: number
  character (LEN=8) :: suit
end type card
type(card), dimension(13) :: suit1,suit0

integer, dimension(2) :: blength=(/1,8/)
integer, dimension(2)::types=(/MPI_INTEGER,MPI_CHARACTER/)
integer, dimension(2) :: delta=(/0,0/)
integer :: intex,MPI_CARD

call MPI_TYPE_EXTENT(MPI_INTEGER,intex,err)
delta(2)=intex
call MPI_TYPE_STRUCT(2,blength,delta,types,MPI_CARD,err)
call MPI_TYPE_COMMIT(MPI_CARD,err)
```

Sample Program (Copying Records)

```
if (rank==1) then
  suit1=card(2,"Clubs") ! All suit1 cards are clubs
  do i=1,13
    suit1(i)%number=i+1
  end do
  call MPI_SEND(suit1,13,MPI_CARD,0,5,MPI_COMM_WORLD,err)

else

  call MPI_RECV(suit0,13,MPI_CARD,1,MPI_ANY_TAG, &
           MPI_COMM_WORLD,status,err)
  print *,"P:",rank," suit0 card array is"
  print *,suit0
end if
```

Program Output

```
P:0  suit0 card array is

2 Clubs 3 Clubs 4 Clubs 5 Clubs 6 Clubs 7 Clubs
8 Clubs 9 Clubs 10 Clubs
11 Clubs 12 Clubs 13 Clubs 14 Clubs
```

KIND Variables

- Q: How can I transfer data between Fortran 90 variables declared to be a specific KIND?
- Programmer must do preliminary work to find out the size (in bytes) of the KIND variables they are using
- This size can then be use to set MPI_Datatype for communication
- Process <u>greatly</u> simplified by use of MPI_TYPE_MATCH_SIZE() routine in MPI-2

Preliminary Program

```
 integer, parameter :: r10=selected_real_kind(10)
real(kind=r10)::a
real(kind=KIND(1.0D0))::b
integer :: bytes
if (rank==0) then
  print *,range(a),precision(a),kind(a)
  call MPI_TYPE_EXTENT(MPI_REAL,bytes,err)
  PRINT *, "kind=",kind(REAL), "r_extent=", bytes
  call MPI_TYPE_EXTENT(MPI_DOUBLE_PRECISION,bytes,err)
  PRINT *, "kind=",kind(b), "dp_extent=", bytes
end if
-----------------------------------------------------
```

```
307  15          8
kind=4 r_extent=4
kind=8 dp_extent=8
```

Sample Program

```
  integer, parameter :: r10=selected_real_kind(10)
  real(kind=r10)::x(1),y(1)
  if (rank == 0) then
    x(1)=exp(1.0)
    write(6,30)x(1)
30  format("x=",f15.13)
    call MPI_SEND(x,1,MPI_REAL8,1,810,MPI_COMM_WORLD,err)
  else
call MPI_RECV(y,1,MPI_REAL8,0,810,MPI_COMM_WORLD,info,err)
    write(6,20) y(1)
20  format("y=",f15.13)
  end if
--------------------------------------------------
x=2.7182818284590
y=2.7182818284590
```

Allocatable Arrays

- Q: "Can an MPI process create an allocatable array and then duplicate it exactly in another process' memory?"
- Yes, but the sending process has two provide size information to the receiver before the actual data transfer is made.

Sample Program

```
   real, dimension(:,:), allocatable ::data,value
   integer, dimension(2) :: dim
   if (rank==1) then
     read (10,*)row,col ! Reads in 5 rows 8 columns
     allocate(data(row,col))
     dim=shape(data)
     call MPI_SEND(dim,2,MPI_INTEGER,0,77,MPI_COMM_WORLD,err)
     data=reshape( (/ ((real(i*j),i=1,row),j=1,col) /) &
           ,SHAPE=(/row,col/))
     call MPI_SEND(data,row*col,MPI_REAL,0,55,MPI_COMM_WORLD,err)
   else
 call MPI_RECV(dim,2,MPI_INTEGER,1,77,MPI_COMM_WORLD, status,err)
     allocate(value(dim(1),dim(2)))
 call MPI_RECV(value,dim(1)*dim(2),MPI_REAL,1,55,MPI_COMM_WORLD,&
                  status,err)
     print *,"value array is:"
     print *,value
   end if
```

Program Output

```
value array is:
  1.000000       2.000000       3.000000       4.000000
  5.000000       6.000000       7.000000       8.000000

  2.000000       4.000000       6.000000       8.000000
  10.00000       12.00000       14.00000       16.00000

  3.000000       6.000000       9.000000       12.00000
  15.00000       18.00000       21.00000       24.00000

  4.000000       8.000000       12.00000       16.00000
  20.00000       24.00000       28.00000       32.00000

  5.000000       10.00000       15.00000       20.00000
  25.00000       30.00000       35.00000       40.00000
```

UK - BU - OSC - Alliance - NCSA - HPCERC - PACS Training
Asynchronous MPI Training

http://webct.ncsa.uiuc.edu:8900/

Array Sections

- Q:"Can a section of an existing array be used explicitly in MPI-1 communication routines?"
- Yes, the array section notation is recognized (and converted)
- In the sample program, we will transfer 3 elements of a 10 element array *data*. The elements are *data(3), data(5), and data(7)*

Sample Program

```
real :: data(10),part(7)
…
if (rank==1) then
  data=(/ (REAL(i)**2,i=1,10) /)
call MPI_SEND(data(3:7:2),3,MPI_REAL,0,11,MPI_COMM_WORLD,&
                                                      err
else
  call MPI_RECV(part,7,MPI_REAL,1,11,MPI_COMM_WORLD, &
      status,err)
  call MPI_GET_COUNT(status,MPI_REAL,count,err)
  print *,"P:",rank," got ",count," elements"
  do i=1,7
    print *,"P:",rank," part index value=",i,part(i)
  end do
end if
```

Program Output

```
P:0  got 3 elements
P:0  part index value= 1   9.000000
P:0  part index value= 2   25.00000
P:0  part index value= 3   49.00000
P:0  part index value= 4   0.000000E+00
P:0  part index value= 5   0.000000E+00
P:0  part index value= 6   0.000000E+00
P:0  part index value= 7   0.000000E+00
```

Array Indexing

- Q:"Is non-default array indexing allowed in MPI-1 Message Passing?"
- A: Yes, whatever index value range is chosen by the user will will be recognized both is MPI_SEND() and MPI_RECV()
- In our program an array with "axes-dimensioning" is sent to a Fortran array with "C indexing"

http://webct.ncsa.uiuc.edu:8900/

Sample Program

```
real :: data(-5:5),part(0:9)
…
if (rank==1) then
  data=(/ (REAL(i)**2,i=-5,5) /)

call MPI_SEND(data,10,MPI_REAL,0,1111,MPI_COMM_WORLD,err)
else
 call MPI_RECV(part,10,MPI_REAL,1,1111,MPI_COMM_WORLD, &
      status,err)
  call MPI_GET_COUNT(status,MPI_REAL,count,err)
  print *,"P:",rank," got ",count," elements"

  do i=0,9
    print *,"P:",rank," part index value=",i,part(i)
  end do

end if
```

Program Output

```
P:0  got 10  elements
P:0  part index value= 0   25.00000
P:0  part index value= 1   16.00000
P:0  part index value= 2   9.000000
P:0  part index value= 3   4.000000
P:0  part index value= 4   1.000000
P:0  part index value= 5   0.000000E+00
P:0  part index value= 6   1.000000
P:0  part index value= 7   4.000000
P:0  part index value= 8   9.000000
P:0  part index value= 9   16.00000
```

Pointers and Targets

- Q:"Can Fortran 90 pointer data be transferred in MPI-1 Message Passing?"
- Yes, relatively simply
- It is interesting to see what happens to the targets the transferred pointers are associated with.
- The changes in the targets is both straightforward and disturbing at the same time …

Sample Program

```
real,pointer :: p0,p1
real,target :: t0,t1

if (rank==0) then
  t0=19.0
  p0=>t0
  call MPI_SEND(p0,1,MPI_REAL,1,55,MPI_COMM_WORLD,err)
else
  t1=52.0
  p1=>t1
  print *,"P:",rank,"t1 before recv",t1
  call MPI_RECV(p1,1,MPI_REAL,MPI_ANY_SOURCE,55, &
              MPI_COMM_WORLD,status,err)
  print *,"P:",rank,"t1 after recv",t1
end if
```

Program Output

```
P:1 t1 before recv   52.00000
P:1 t1 after recv   19.00000
```

Notable Feature: Process 0 changed the value of *t1* without actually sending a message to it, but by going through its pointer *p1*.

C++ and MPI-1

- Pointer Variable Transfer
- Reference Variable Transfer
- Structures: Byte Stream Transfer
- Object Transfer
- Transferring Inheritance traits
- MPI-2 C++ Bindings Namespace

Pointer Message Passing

- Goal is to pass a C/C++ pointer variable containing an address to another pointer variable in another MPI processor's memory
- The receiving process should then be able to indirectly access contents of actual variable "pointed to"
- Difficult to accomplish since a memory address in one MPI process' memory could mean nothing to a different MPI process' memory.
- Technique shown in following program insures compatible addressing by using static array variables
- Since there is no MPI_Datatype for pointer variables had to determine the address size (using C sizeof operator) and transfer the pointer as a byte stream.

Pointer C Progam

```
#include <mpi.h>
#include <stdio.h>
int main(int argc,char** argv) {
   static double x[2]={89.32,156.2};
   double *dptr[2];
   int rank,i;
   MPI_Status state;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if (rank==0) {
     MPI_Address(&x[0],(MPI_Aint*)&dptr[0]);
     MPI_Address(&x[1],(MPI_Aint*)&dptr[1]);
     printf("P:%d pointer value=%X\n",rank,dptr[0]);
     printf("P:%d pointer contents=%f\n",rank,*dptr[0]);
     printf("P:%d pointer value=%X\n",rank,dptr[1]);
     printf("P:%d pointer contents=%f\n",rank,*dptr[1]);

     MPI_Send(dptr,8,MPI_BYTE,1,23,MPI_COMM_WORLD);
   }else{
```

Pointer Program: Reception

```
    MPI_Recv(dptr,8,MPI_BYTE,0,23,MPI_COMM_WORLD,&state);

    printf("P:%d pointer value=%X\n",rank,dptr[0]);
    printf("P:%d pointer contents=%f\n",rank,*dptr[0]);

    printf("P:%d pointer value=%X\n",rank,dptr[1]);
    printf("P:%d pointer contents=%f\n",rank,*dptr[1]);

    *dptr[1]=11.11;
  }

  printf("P:%d x[1]=%f\n",rank,x[1]);

  MPI_Finalize();

 return 0;
}
```

http://webct.ncsa.uiuc.edu:8900/

Pointer Program Output

```
P:0 pointer value=808D258      // Before Send
P:0 pointer contents=89.320000
P:0 pointer value=808D260
P:0 pointer contents=156.200000

P:1 pointer value=808D258   // After Recv
P:1 pointer contents=89.320000
P:1 pointer value=808D260
P:1 pointer contents=156.200000

P:0 x[1]=156.200000 // After if-else
P:1 x[1]=11.110000
```

Reference Variables

- Reference variables are a C++ addition to the combined languages in which one memory location can be given two names.
- The name of the declared variable is one, the name of a reference variable assigned to the "real" name is the other
- Put into the language to allow direct, readable transfer-by-reference between dummy arguments and actual arguments
- Even though there is no reference variable MPI_Datatype, the following code shows how the contents of a reference variable can be transferred between two MPI processes.

Reference Program

```
#include <mpi.h>
#include <stdio.h>
int main(int argc,char** argv) {
   double x;
   double& y=x;
   int rank;
   MPI_Status state;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if (rank==0) {
     x=3.78;
     printf("P:%d x=%f\n",rank,x);
     printf("P:%d y=%f\n",rank,y);
     MPI_Send(&y,8,MPI_BYTE,1,33,MPI_COMM_WORLD);
   } else {
     MPI_Recv(&y,8,MPI_BYTE,0,33,MPI_COMM_WORLD,&state);
     printf("P:%d x=%f\n",rank,x);
     printf("P:%d y=%f\n",rank,y);
     y=8.7;
   }
   printf("P:%d x=%f\n",rank,x);
   printf("P:%d y=%f\n",rank,y);
   MPI_Finalize(); return 0;}
```

Reference Program Output

```
P:0 x=3.780000 // Before Send
P:0 y=3.780000

P:1 x=3.780000 // After Recv
P:1 y=3.780000

P:0 x=3.780000 // After if-else
P:0 y=3.780000
P:1 x=8.700000
P:1 y=8.700000
```

Structures as Byte Streams

- As you were taught in an introductory MPI course the easiest and clearest method for transferring a structure was to make your own new, derived MPI_Datatype that matched the memory layout of the structure.
- An alternate (less elegant) approach is to just pass each data item in the C structure one-by-one as a stream of bytes.
- Both approaches DO the same thing: transfer the memory map of the structure variable
- The following code shows this "new" method

Structure Program

```
#include <mpi.h>
#include <stdio.h>
struct particle{
  int id;
  double x,y,z;
};
int main(int argc,char** argv) {
   static struct particle atom={3492,3.45,-0.58,13.26};
   int rank,count;
   MPI_Status sitrep;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if (rank==0) {
     printf("P:%d atom id =%d\n",rank,atom.id);
     printf("P:%d atom position %f %f %f\n",rank,atom.x,
                    atom.y,atom.z);
     MPI_Send(&atom,28,MPI_BYTE,1,23,MPI_COMM_WORLD);
   }else{
```

http://webct.ncsa.uiuc.edu:8900/

Structure Reception

```
    MPI_Recv(&atom,28,MPI_BYTE,0,23,MPI_COMM_WORLD,
                                  &sitrep);
    printf("P:%d atom id =%d\n",rank,atom.id);
    printf("P:%d atom position %f %f %f\n",rank,atom.x,
                             atom.y,atom.z);

    MPI_Get_count(&sitrep,MPI_BYTE,&count);
    printf("P:%d recv byte count is %d\n",rank,count);
  }

 MPI_Finalize();

 return 0;
}
```

Structure Program Output

```
P:0 atom id =3492  // Before Send
P:0 atom position 3.450000 -0.580000 13.260000

P:1 atom id =3492  // After Recv
P:1 atom position 3.450000 -0.580000 13.260000

P:1 recv byte count is 28 // Just checking …
```

Object Messaging Passing

- By far the most asked question from C++ programmers. How can I send an object from one process to another? (Using only MPI –1 routines)
- Answer: just send the data members
- When a Class is defined every object created knows the address (in its local memory) of the function members through a look-up table.
- When a specific object calls a member function that address is accessed and the "this" pointer for that object is passed as an argument
- The key is that the address table is known to both the sending and receiving processes.
- In the following code and object is transferred as a byte data stream

http://webct.ncsa.uiuc.edu:8900/

Object Definition

```
#include <mpi.h>
#include <iostream.h>

class Triangle {
      double base;
      double height;
  public:
      void set(double a,double b) {
          base=a; height=b;
      }
      void display() {
        cout << "base=" << base << endl;
        cout << "height=" << height << endl;
      }
      double area() { return (0.5*base*height);}
};
```

Object Transfer

```
int main(int argc,char** argv) {
   Triangle t;
   Triangle s;
   int rank;
   MPI_Status sitrep;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if (rank==0) {
     t.set(12.0,9.25);
     t.display();
     cout << "P:" << rank << " area=" << t.area() << endl;
     MPI_Send(&t,16,MPI_BYTE,1,23,MPI_COMM_WORLD);
   }else{
     MPI_Recv(&s,16,MPI_BYTE,0,23,MPI_COMM_WORLD,&sitrep);
     s.display();
     cout << "P:" << rank << " area=" << s.area() << endl;
   }
   MPI_Finalize();
   return 0; }
```

Object Program Output

```
base=12        // Before Send, from Process 0
height=9.25
P:0 area=55.5

base=12        // After Recv, from Process 1
height=9.25
P:1 area=55.5
```

Object “Datatype”

- Since (as shown) only the data members of an object need to be transferred, one can create a new derived MPI_Datatype for the memory layout of the data members.
- The derived MPI_Datatype can then be used in the message passing as an alternative to a byte stream.
- In the following code, the new MPI_Datatype called MPI_TRI is created for use with Triangle objects. The output is the same as the previous program.

MPI_TRI Program

```
int main(int argc,char** argv) {
   Triangle t,s;
   int rank;
   MPI_Status sitrep;
   MPI_Datatype MPI_TRI;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Type_contiguous(2,MPI_DOUBLE,&MPI_TRI);
   MPI_Type_commit(&MPI_TRI);
   if (rank==0) {
     t.set(12.0,9.25);
     t.display();
     cout << "P:" << rank << " area=" << t.area() << endl;
     MPI_Send(&t,1,MPI_TRI,1,23,MPI_COMM_WORLD);
   }else{
     MPI_Recv(&s,1,MPI_TRI,0,23,MPI_COMM_WORLD,&sitrep);
     s.display();
     cout << "P:" << rank << " area=" << s.area() << endl;
   }
   MPI_Finalize(); return 0; }
```

Inheritance

- We have just seen two methods for transferring objects, but with if the object is a member of a derived class?
- Programmers would like the received object to also inherit the capabilities of the base class.
- This will be accomplished if ALL of the data members in the inheritance "lineage" are transferred.
- In the following program, there is a base class and one derived class; an object of the derived class will be sent and received

Inheritance Base Class

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

const int N=9;

class Data {
  protected:
    double x[N],y[N];
  public:
     void getdata() {
       FILE *dfile;
       dfile=fopen("data.in","r");
       for (int i=0;i<N;++i)
          fscanf(dfile,"%lf%lf",&x[i],&y[i]);
       fclose(dfile);
     }
     void display() {
       for (int i=0;i<N;++i)
        cout << "i=" << i << " x=" << x[i] << " y=" << y[i] <<endl;
     } };
```

Inheritance Derived Class

```
class Average: public Data {
    double mux,muy;
  public:
    void averagex() {
      double sum=0.0;
      for (int i=0; i<N; ++i)
         sum += x[i];
      mux=sum/N; }
void averagey() {
      double sum=0.0;
      for (int i=0; i<N; ++i)
         sum += y[i];
      muy=sum/N;
    }
    void display() {
      cout << "Average x: " << mux << endl;
      cout << "Average y: " << muy << endl;
    }
};
```

Inheritance MPI code

```
int main(int argc,char** argv) {
   Average temp,heat;
   int rank;
   MPI_Status sitrep;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if (rank==0) {
     temp.getdata();
     temp.averagex();
     temp.averagey();
     MPI_Send(&temp,160,MPI_BYTE,1,23,MPI_COMM_WORLD);
   }else{
     MPI_Recv(&heat,160,MPI_BYTE,0,23,MPI_COMM_WORLD,&sitrep);
     heat.Data::display();
     heat.display();
   }
   MPI_Finalize();
   return 0; }
```

Inheritance Code Output

```
i=0 x=1 y=15.6  // Data::display() from P1
i=1 x=2 y=17.5
i=2 x=3 y=36.6
i=3 x=4 y=43.8
i=4 x=5 y=58.2
i=5 x=6 y=61.6
i=6 x=7 y=64.2
i=7 x=8 y=70.4
i=8 x=9 y=98.8

Average x: 5    // Average::Display() from P1
Average y: 51.8556
```

MPI-2 C++ Bindings

- Some MPI libraries have incorporated the MPI-2 feature of providing C++ bindings for MPI functions.
- The MPI-2 strategy is to use a new namespace called MPI and put a number of high-level Class definitions in it.
- Each MPI namespace Class is required to have a default constructor, a destructor, a copy constructor, and assignment operator

MPI Namespace

```
namespace MPI {
  class Comm                                    {...};
  class Intracomm : public Comm                 {...};
  class Graphcomm : public Intracomm            {...};
  class Cartcomm  : public Intracomm            {...};
  class Intercomm : public Comm                 {...};
  class Datatype                                {...};
  class Errhandler                              {...};
  class Exception                               {...};
  class Group                                   {...};
  class Op                                      {...};
  class Request                                 {...};
  class Prequest  : public Request              {...};
  class Status                                  {...};
};
```

Class Member Functions

- The actual C++ version of the MPI functions you want to use in your program are member functions of one of the MPI namespace classes.
- Thus, the general syntax to reference a C++ MPI function is

  MPI::<class object>.<mpi function>

- As an example, the C++ version of the broadcast function for the "World" communicator would be used with the following syntax

  MPI::COMM_WORLD.Bcast(data, 100, MPI::INT, 5);

- As evident in this example, the symbolic names for MPI constants have also changed in the C++ version

Point-to-Point Program

- On the next page is shown the "classic" beginning MPI program in which one MPI process sends data to another.
- But, it has been written entirely with MPI-2 C++ versions of the necessary functions and constants.
- Reference: The MPI-2 Standard Document

P2P C++ Program (Send)

```
#include <mpi.h>
#include <iostream.h>

int main(int argc, char *argv[]) {
  int rank,size;
  double num[50],data[500];
  MPI::Status Sitrep;

  MPI::Init(argc,argv);
  size=MPI::COMM_WORLD.Get_size();
  rank=MPI::COMM_WORLD.Get_rank();

  if (rank==0) {
    for (int i=0;i<50;++i) {
       num[i]=i+size;
    }
    MPI::COMM_WORLD.Send(num,50,MPI::DOUBLE,1,45);
```

http://webct.ncsa.uiuc.edu:8900/

P2P C++ Program (Receive)

```
}else{
    MPI::COMM_WORLD.Recv(data,500,MPI::DOUBLE,
                  MPI::ANY_SOURCE,MPI::ANY_TAG,Sitrep);

    cout << "P:" << rank  << " source=" <<
         Sitrep.Get_source() << endl;
    cout << "P:" << rank  << " tag=" << Sitrep.Get_tag()
         << endl;
    int count=Sitrep.Get_count(MPI::DOUBLE);
    cout << "P:" << rank  << " count="  << count << endl;

    for (int i=0;i<count;i+=10) {
     cout << "i=" << i << " data[i]=" << data[i] << endl;
    }
  }
MPI::Finalize(); return 0; }
```

http://webct.ncsa.uiuc.edu:8900/

Problem Set

1) In this MPI progam, Process 0 will have a normal array ,that is, a statically declared array of reals. The programmer is free to pick the dimensions and calculate the elements in whatever way they like. Process 1 has an allocatable array. Write MPI Fortran 90 code that transfers the array from Process 0 into the array of Process1. You must work under the assumption that Process 1 knows absolutely nothing about the Process 0 array (except its dimensionality).

   Have your code output whatever is necessary to confirm that it worked as expected

Problem Set

2) Write C++ code that will transfer a derived object from one process to another. The base class should be called **Rect**. It should have two data members, length and width and member functions will give values to length and width, calculate the area of the rectangle, calculate the perimeter of the rectangle, and print out the area and perimeter.

A class called **Zoid** will be the derived class. It should have one data member which is the height of a rectangular column. It should have member functions which initialize the height, calculate the column volume, and print out its volume.

Problem Set

Process0 should send a **Zoid** object to Process1. Before sending it should initialize the length and width of the **Rect** class. After reception, Process1 should initialize the height, and then should output all three calculated values: perimeter, area, and volume.