INTRODUCTION TO MPI
A Tutorial with Exercises

By: Rosalinda de Fainchtein, Ph.D.
CSC/NASA GSFC, Code 931

- This tutorial is adapted from a class by the same name taught as a service to the NCCS User Community.
- As you follow this tutorial, you will write simple MPI parallel programs, and learn some of the nuances of MPI.
- Each topic includes a simple exercise for you to apply the material learned. A solution for each of these exercises is also provided.
- While the material can be scanned rather quickly, doing the simple exercises should help you derive the most benefit from this tutorial.
- The material targets students with no prior parallel programming experience, who know Fortran. If you are a C programmer, you should be able to follow the examples, nonetheless.
- A subdirectory containing C versions of all the examples and

exercises is also available through either anonymous ftp from UniTree, or to be copied from "jsimpson" (at /scr/mpi-class/C) by those who have access to this machine.

CONTENTS

What is MPI?

MPI is a library of subroutines for handling communication and synchronization for programs running on parallel platforms.

- MPI targets distributed memory platforms, such as the Cray T3E, but it often delivers improved performance on shared memory platforms also (such as the SGI Origin).

- MPI is portable.

- MPI programs usually follow a single program multiple data (SPMD) format.

How do I Run My MPI Program?

Given a program MY_MPI_PROGRAM.f where MPI is initialized and used:

- Compile as usual

- f90 -o MY_MPI_PROGRAM MY_MPI_PROGRAM.f

- Run the executable using *m* processors:

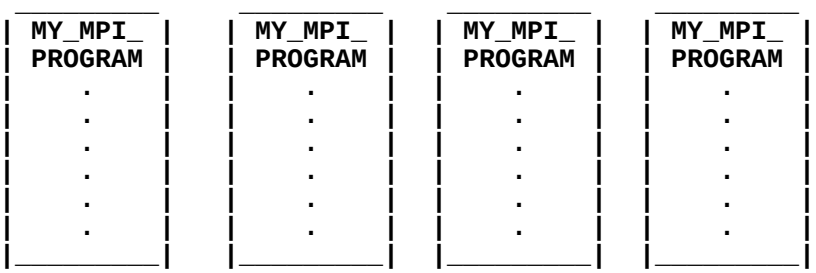
  mpirun -np *m* MY_MPI_PROGRAM

How does MPI get MY_MPI_PROGRAM to run in parallel?

MPI spawns an identical copy of MY_MPI_PROGRAM on each of

the m requested processors.

e.g. If *m*=4, there will be four identical jobs running on four processors at the same time!



Example 1: ("HELLO WORLD")

```
      program example1

      implicit none

!--Include the mpi header file
      include 'mpif.h'
      integer ierr,myid,numprocs
      integer irc

!--Initialize MPI
      call MPI_INIT( ierr )
```

```
!--Who am I? --- get my rank=myid
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )

!--How many processes in the global group?
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      print *, "Process ",myid," of ",numprocs," is alive"

!--Finalize MPI
      call MPI_FINALIZE(irc)

      stop
      end
```

Exercise 1

---

1. Copy Example 1 from jsimpson at: /scr/mpi-class/example1.f into your directory.

   (or use anonymous ftp to UniTree (dirac) to retrieve example1.f)

2. Compile the code and run it on 4 processors.

Output from example1:

```
jsimpson% f90 -o example1 example1.f
jsimpson% mpirun -np 4 example1
Process  0  of  4  is alive
Process  2  of  4  is alive
Process  1  of  4  is alive
Process  3  of  4  is alive
STOP (PE 3)   executed at line 24 in Fortran routine 'EXAMPLE1'
```

```
STOP (PE 0)   executed at line 24 in Fortran routine 'EXAMPLE1'
STOP (PE 2)   executed at line 24 in Fortran routine 'EXAMPLE1'
STOP (PE 1)   executed at line 24 in Fortran routine 'EXAMPLE1'
```


HOW do I assign different work to each processor, if all processors run the same program?

The short answer:

- *MPI assigns a* **rank** *(an integer number) to each process to identify it.*
- *The routine* **MPI_COM_RANK** *returns the rank of the calling process.*
- *The routine* **MPI_COM_SIZE** *returns the size or number of processes in the application.*

These two parameters, size and rank, can then be used (through block if statements or otherwise) to differentiate the computations that each process will execute.

```
if (my_rank == 0) then
x= ....
y= ....
end if
if (my_rank == size-1) then
z= ...
```

```
.....
end if
```


The Longer Answer:

When MPI is initialized, it creates a **communicator**, consisting of a **group** of processes and their labels. This communicator is called,

MPI_COMM_WORLD

The number of processes (m) in this communicator is determined when we submit the MPI job:

mpirun -np m MY_MPI_JOB

Each process can probe for the value of m by calling the MPI routine

MPI_COMM_SIZE

Each process in the MPI_COMM_WORLD group is assigned a rank, an integer with incremental value between 0 and m-1. Each process can determine its own rank by calling the routine

MPI_COMM_RANK

What are the minimum entries in a program to run an MPI job?

There are three required entries on any MPI program:

- **include ’mpif.h’**
- **call MPI_INIT(ierr)**
- **call MPI_FINALIZE(ierr)**

(No MPI program will run without these statements!)

A template for an MPI program can be found at jsimpson in

/scr/mpi-class/template.f

(or use anonymous ftp to UniTree to retrieve template.f)

Template for an MPI program

```
   program template

!-- Template for any mpi program

   implicit none    ! highly recommended. It will make
                    ! debugging infinitely easier.

!--Include the mpi header file
   include 'mpif.h'           ! --> Required statement

!--Declare all variables and arrays.
   integer ierr,myid,numprocs,itag
   integer irc

!--Initialize MPI
   call MPI_INIT( ierr )      ! --> Required statement

!--Who am I? --- get my rank=myid
   call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )

!--How many processes in the global group?
   call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

!--Finalize MPI
   call MPI_FINALIZE(irc)     ! ---> Required statement

   stop
   end
```

Exercise 2

Starting from template.f, write a program that given a common value of x (e.g. x=5 in all processes), computes:

- $y=x^2$ in process 0
- $y=x^3$ in process 1
- $y=x^4$ in process 2

and writes a statement from each process that identifies the process and reports the values of x and y from that process.

Solution to Exercise 2

(See the full program at /scr/mpi-class/exercise2-I.f in jsimpson, or use anonymous ftp to UniTree to retrieve exercise2-I.f)

```
program template
.
```

```
.
!--Define new variables used
   integer ip
   real x,y
      .
      .
   {MPI_COMM_RANK and MPI_COMM_SIZE CALLS......}

!--Insert the calculations after the size (numprocs)
!  and rank (myid) are known.

!--Set the value of x on all processes
   x=5.

!--Define the value of y on each process .....

     if(myid == 0) then
      y=x**2
     else if (myid == 1) then
      y=x**3
     else if (myid == 2) then
      y=x**4
     end if


!--...and print it.

     write(*,*)'On process ',myid,' y=',y

      .
      .
   stop
   end
```

(A more concise version can be found in /scr/mpi-class/exercise2-I.f in jsimpson, or use anonymous ftp to UniTree to retrieve exercise2-II.f)

```
program template
.
.
```

```
!--Define new variables used
   integer ip
   real x,y
      .
      .
   {MPI_COMM_RANK and MPI_COMM_SIZE CALLS......}

!--Insert the calculations after the size (numprocs)
!  and rank (myid) are known.

!--Set the value of x on all processes
   x=5.

!--Define the value of y on each process and print it.

        y=x**(2+myid)
        write(*,*)'On process ',myid,' y=',y

      .
      .
   stop
   end
```


Output from exercise 2:

```
jsimpson% mpirun -np 3 exercise2
On process  2  y= 625.
On process  0  y= 25.
On process  1  y= 125.
STOP (PE 1)   executed at line 40 in Fortran routine 'TEMPLATE'
STOP (PE 2)   executed at line 40 in Fortran routine 'TEMPLATE'
STOP (PE 0)   executed at line 40 in Fortran routine 'TEMPLATE'
```

COMMUNICATIONS

MPI is designed to manage codes running on distributed memory platforms. Thus data residing on other processes is accessed through MPI calls.

Although MPI includes a large number of communication routines, most applications require only a handful of them.
A minimal set of routines that most parallel codes run with are:

- MPI_INIT
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECV
- MPI_FINALIZE

POINT TO POINT COMMUNICATIONS

- MPI_SEND and MPI_RECV perform "point to point" communications.

- The two routines work together to complete a transfer of data from one process to another.

- One process posts a send operation, and the target process posts a receive for the databeing transferred.

e.g. (pseudocode)

```
if (my_rank == 0)
&   call MPI_SEND(......,1,..)

if (my_rank == 1)
&   call MPI_RECV(......,0,.....)
```


Blocking MPI Send Routine

MPI_SEND(buf,count,datatype,dest,tag,comm,ie

- **buf** =initial address of send buffer (choice)
- **count** =number of entries to send (integer)
- **datatype**=datatype of each message entry (handle)
- **dest** =rank of destination (integer)
- **tag** =message tag
- **comm** =communicator (handle)
- **ierror** =return error code (integer)

datatype available handles:

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

Blocking MPI Receive Routine

MPI_RECV(buf,count,datatype,source,tag,comm

- **buf** =initial address of send buffer (choice)
- **count** =number of entries to send (integer)
- **datatype**=datatype of each entry (handle)
- **source** =rank of source (integer)
- **tag** =message tag
- **comm** =communicator (handle)
- **status** =return status array(integer)
- **ierror** =return error code (integer)

Any routine that calls MPI_RECV, should declare the status array:

integer status(MPI_STATUS_SIZE)

The status array contains information on the received message, such as its tag, source, error code. The number of entries received can also be obtained from this array.

EXAMPLE 2: Point to Point Communication

Process 0 sends the array x to process 1:

```
      .
      .
      integer count,dest,tag,ierror
      integer status(MPI_STATUS_SIZE)
      real x(5),y(5)
      .
      .
! --the common calling arguments
      count   =5   ---->! 5 buffers (variables) to be transferred
      datatype=MPI_REAL
      tag     = 2
      comm    = MPI_COMM_WORLD

!---process 0 send data to process 1 (dest=1)
      if (my_rank == 0) then
      dest    = 1
      call MPI_SEND(x(1),count,datatype,
      &        dest,tag,comm,          ierror)
! --------------> [Note: x(1) is the first of "count" buffers to be sent]
      end if

!---process 1 receives data from process 0 (source=0)
      if (my_rank == 1) then
        source=0
        call MPI_RECV(y(1),count,datatype,
      &               source,tag,comm,status,  ierror)
      end if
```

The full program can be copied from jsimpson at /scr/mpi-class/example2.f, or by anonymous ftp to UniTree.

THE MESSAGE ENVELOPE

How does an MPI process select which message to receive if more than one message has been sent to it?

Each message carries verification information with it called the **message** envelope.

The message envelope consists of the following:

- **source**
- **destination**
- **tag**
- **communicator**

A message is received only if the arguments in the posted receive call agree with the message envelope of an incoming message.

Exercise 3

Using the program in exercise 2, send all values of y to process 0 and

compute the average of y at process 0.
Print the result, including the process rank from where it is being printed.

Solution of Exercise 3

```
      program template
        .
        .
      integer tag,status(MPI_STATUS_SIZE)
      real x,y,buff
        .
        .
      {CALCULATION OF Y ON THE DIFFT. PROCESSES}
        .
!--Average the values of y
      tag=1
      if (myid == 0)then
      do ip=1,numprocs-1
        call MPI_RECV(buff,1,MPI_REAL,ip,tag,
      &                 MPI_COMM_WORLD,status,ierr)
        y=y+buff
      end do
      y=y/float(numprocs)
      write(*,*)'The average value of y is ',y
      else
      call MPI_SEND(y  ,1,MPI_REAL,0,tag,
      &               MPI_COMM_WORLD,         ierr)
      end if
        .
        .
      stop
      end
```

The full program can be found in jsimpson at:
/scr/mpi-class/exercise3.f, or by anonymous ftp to UniTree.

Output from exercise 3

```
mpirun -np 3 exercise3
On process  0  y= 25.
On process  2  y= 625.
On process  1  y= 125.
The average value of y is  258.33333333333331
STOP (PE 1)   executed at line 56 in Fortran routine 'EXERCISE3'
STOP (PE 2)   executed at line 56 in Fortran routine 'EXERCISE3'
STOP (PE 0)   executed at line 56 in Fortran routine 'EXERCISE3'
```


WILDCARDS

What if I want to receive a message regardless of its source and/or tag?
====> Replace the source and/or tag entry on the MPI_RECV call with a **wildcard**:

- **Ignore source** by using the **MPI_ANY_SOURCE** wildcard:

```
      call MPI_RECV(buf,count,datatype,
     &     MPI_ANY_SOURCE,tag,comm,status, ierror)
```

- **Ignore tag** by using the **MPI_ANY_TAG** wildcard:

```
      call MPI_RECV(buf,count,datatype,
     &     source,MPI_ANY_TAG,comm,status, ierror)
```

- **Ignore tag and source**:

```
      call MPI_RECV(buf,count,datatype,
     &     MPI_ANY_SOURCE,MPI_ANY_TAG,comm,status, ierror)
```

WILDCARDS SHOULD ONLY BE USED WHEN ABSOLUTELY NECESARY!

Blocking vs Non-Blocking Communications

MPI_SEND and MPI_RECV are **blocking** communications routines.

What does it mean for MPI_SEND to be a blocking routine?

Once a call to MPI_SEND is posted by a process, the call does

not return control to the calling program or routine, until the buffer containing the data to be copied unto the receiving process can be safely overwritten (This insures that the message being sent is not "corrupted" before the sending is complete)

What does it mean for MPI_RECV to be a blocking routine?

The call does not return control to the calling program until the data to be received has in fact been received.

Avoiding "Hung" Processes

A "**Hung**" condition occurs when one or more processes reach a call to an MPI block_receive routine, but the message never arrives. The process will wait indefinitely and no error message will be generated. (The same will happen with a block send message that is never completed).

A "**Hung**" condition can occur when two processes exchange messages, if the exchange is not programmed carefully.

First: an example of an exchange that will never "hang":
Can you tell why? (If not compare to the example on next page).

```
!--Exchange messages
     if (myid == 0) then
     call mpi_send(a,1,mpi_real,1,tag,MPI_COMM_WORLD,ierr)
     call mpi_recv(b,1,mpi_real,1,tag,MPI_COMM_WORLD,
     &              status,ierr)
     elseif (myid == 1) then
```

```
      call mpi_recv(a,1,mpi_real,0,tag,MPI_COMM_WORLD,
      &              status,ierr)
      call mpi_send(b,1,mpi_real,0,tag,MPI_COMM_WORLD,ierr)
      end if
```

The code above is an excerpt from example3_a.f found in jsimpson at /scr/mpi-class/example3_a.f. It can also be downloaded by anonymous ftp to UniTree.

(you can verify that this code will run to completion without a problem).

Example of a hanging program

Suppose that the order of the send and receive calls are modified as follows

```
!--Exchange messages
      if (myid == 0) then
        call mpi_recv(b,1,mpi_real,1,tag,MPI_COMM_WORLD,
      &               status,ierr)
        call mpi_send(a,1,mpi_real,1,tag,MPI_COMM_WORLD,ie
      elseif (myid == 1) then
        call mpi_recv(a,1,mpi_real,0,tag,MPI_COMM_WORLD,
      &               status,ierr)
        call mpi_send(b,1,mpi_real,0,tag,MPI_COMM_WORLD,ie
      end if
```

The code above is an excerpt from example3_a.f found at /scr/mpi-class/example3_c.f. It can also be downloaded by anonymous ftp to UniTree.

Will this program run as well as example3_a? Why?

Example of a hanging program: Why?

The program in example3_c.f will not run at all, **it will hang!** (that is, it will never complete and give no error diagnostic -- other that running out of time).

Here is why:

1. Each of processes 0 and 1 calls a blocking MPI_RECV routine and expects to receive a message from the other process.

2. Neither process will continue on to the next statement until the information has been received (or is at least safely on its way).

3. At this point neither process has actually SENT any message

to the other process.

4. **Thus both processes will wait indefinitely for a message that will never come....**

Example of a Program that MIGHT Hang

The following order of the send and receive calls will work on some platforms but not others.

IT IS NOT RECOMMENDED!

```
!--Exchange messages
     if (myid == 0) then
      call mpi_send(a,1,mpi_real,1,tag,MPI_COMM_WORLD,ier
      call mpi_recv(b,1,mpi_real,1,tag,MPI_COMM_WORLD,
     &              status,ierr)
     elseif (myid == 1) then
      call mpi_send(b,1,mpi_real,0,tag,MPI_COMM_WORLD,ier
      call mpi_recv(a,1,mpi_real,0,tag,MPI_COMM_WORLD,
     &              status,ierr)
     end if
```

The full program can be found in jsimpson at: /scr/mpi-class/example3_b.f, or by anonymous ftp to UniTree.

A safe alternative is to use the MPI_SENDRECV routine instead.

```
      MPI_SENDRECV(sendbuf,sendc
     &             dest,sendtag,recvbuf,recvcount,recvtyp
     &             source,recvtag,comm,status,ierror)
```

- **sendbuf =initial address of send buffer (choice)**
- **sendcount = # of entries to send (integer)**
- **sendtype =type of entries in send buffer (handle)**
- **dest =rank of destination (integer)**
- **sendtag =send tag (integer)**
- **recvbuf =initial address of receive buffer (choice)**
- **recvcount=max. num. of entries to receive (integer)**
- **recvtype =type of entries in receive buffer (handle)**
- **source =rank of source (integer)**
- **recvtag =receive tag (integer)**
- **status =return status(integer)**

- **comm =communicator (handle)**
- **ierror =return error code (integer)**

Example 4: Using MPI_SENDRECV I

Example 4 shows the use of the sendrecv MPI call replacing a pair of consecutive send and receive calls originating from a single process. Note that the communications here are the same as in example3_b, except that the sendrecv insures that no deadlock occurs.

Here is an excerpt from example 4,

```
!--Exchange messages
     tag1=1
     tag2=2
     if (myid == 0) then
      call mpi_sendrecv(a,1,mpi_real,1,tag1,
    &                   b,1,mpi_real,1,tag2,
    &             MPI_COMM_WORLD, status,ierr)
     elseif (myid == 1) then
      call mpi_sendrecv(b,1,mpi_real,0,tag2,
    &                   a,1,mpi_real,0,tag1,
    &              MPI_COMM_WORLD,status,ierr)
     end if
```

(See /scr/mpi-class/example4.f in jsimpson, or
use anonymous ftp to UniTree to retrieve example4.f)

Example 5: Using MPI_SENDRECV II

MPI_SENDRECV is compatible with simple MPI_SEND and MPI_RECV routines. Here is an example that illustrates the point.

Example 5
Process 0 sends a to process 1 and receives b from process 2:

```
 tag1=1
 tag2=2
 if     (myid == 0) then
   call mpi_sendrecv(a,1,mpi_real,1,tag1,
&                    b,1,mpi_real,2,tag2,
&                    MPI_COMM_WORLD, status,
&                    ierr)
 elseif (myid==1) then
       call mpi_recv(a,1,mpi_real,0,tag1,
&                    MPI_COMM_WORLD, status,
&                    ierr)
 elseif (myid==2) then
       call mpi_send(b,1,mpi_real,0,tag2,
&                    MPI_COMM_WORLD,
&                    ierr)
 end if
```

(See /scr/mpi-class/example5.f in jsimpson, or
use anonymous ftp to UniTree to retrieve example5.f)

NON-BLOCKING COMMUNICATIONS

The most common **non-blocking** MPI communication routines are:

MPI_ISEND(buf,count,datatype,dest,tag,comm, request,ierror)

MPI_IRECV(buf,count,datatype,source,tag,comm, request,ierror)

- **buf** =initial address of send buffer (choice)
- **count** =number of entries to send/receive (integer)
- **datatype**=datatype of each entry (handle)
- **dest** =rank of destination process (integer)
- **source** =rank of source process(integer)

- **tag** =message tag
- **comm** =communicator (handle)
- **request** =request handle (handle)
- **ierror** =return error code (integer)

BLOCKING vs NON-BLOCKING

What is the difference between MPI_ISEND and MPI_SEND?

MPI_ISEND returns control to the calling routine immediately after posting the send call, before it is safe to overwrite (or use) the buffer being sent.

(MPI_IRECV and MPI_RECV differ in a similar way)

What is the advantage of using non-blocking communication routines?

Performance can be improved by allowing computations that

do not involve the buffer being sent (or received) to proceed simultaneously with the communication.

How can the communicated buffer be re-used safely?

The MPI_ISEND (and MPI_IRECV) return a handle: the request argument. The MPI_WAIT routine can later be called in order to "complete" the request communication. MPI_WAIT blocks computation until the request in question is complete and it is safe to re-use the buffer.

BLOCKING vs NON-BLOCKING -- Rephrasing:

- The non-blocking routines **MPI_ISEND** and **MPI_IRECV** are similar to their blocking couterparts, **MPI_SEND** and **MPI_RECV**.
- The difference between them is that non-blocking communications return control to the calling routine

BEFORE it is safe to re-use the buffer being sent or received.

- This allows the program to proceed with computations not involving the communication buffer, while the communication completes.

- Before the program is to use the sent/received buffer, a call to **MPI_WAIT** is necessary.

- **MPI_WAIT** is a blocking routine. It does not return control to the calling routine until it is safe to re-use the buffer.

NON_BLOCKING: An Example

(Pseudocode)

1. Post a non-blocking send of variable a.

2. call MPI_ISEND(a,.....,REQUEST1,...)

3. While the communication of a takes place, compute the values of b, c, and d (which do not involve a).

$b=x^2$
$c=y^3$
$d=b+c$

4. Block computation until it is safe to use a again.

   call MPI_WAIT(REQUEST1,status)

5. Use a on the computation of e, modify a, etc.

   e=a+b
   a=d

COLLECTIVE MPI ROUTINES

Collective MPI routines involve simultaneous communications among all the processors in a communicator group.

There are 3 types of collective communications

- Barrier synchronization
- Global communications
  - Broadcast
  - Gather
  - Scatter
- Global Reduction Operations

- sum
- max
- min,
- etc.

BROADCAST ROUTINE

The broadcast MPI routine is one of the most commonly used collective routines.

- The root process broadcasts the data in buffer to all the processes in the communicator.
- All processes must call MPI_BCAST with the same root value.

MPI_BCAST(buffer,count,datatype, root,comm,ierror)

- **buffer** =initial address of buffer (choice)
- **count** =number of entries in buffer (integer)
- **datatype**=datatype of buffer (handle)
- **root** =rank of broadcasting process (integer)
- **comm** =communicator (handle)

- **ierror** =return error code (integer)

Exercise 5: Using Broadcast

Modify the code for exercise 2 so that the value of x is defined ONLY on processor 0. Add the necessary code to broadcast the value of x to all the other processors.

Solution to Exercise 5

```
program template
.
.
!--Define new variables used
   integer ip
   real x,y
       .
       .
   {MPI_COMM_RANK and MPI_COMM_SIZE CALLS.....}

!--Insert the calculations after the size and rank
!  are known.

!--Set the value of x on process 0.
      if (myid == 0)    x=5.

!--Broadcast the value of x to all processes.
       call MPI_BCAST(x,1,MPI_REAL,0,MPI_COMM_WORLD,ierr)
```

```
!--Define the value of y on each process and print it.
   do ip=1,numprocs
   if(myid == ip-1) then
   y=x**(2+myid)
   write(*,*)'On process ',myid,' y=',y
   end if
   end do
      .
      .
   stop
   end
```

(See the full program at /scr/mpi-class/exercise5.f,
or use anonymous ftp to UniTree to retrieve exercise5.f)

REFERENCES

- ***To use as a general MPI reference:***
  M,Snir, et.al, *MPI The Complete Reference*, second edition, MIT Press 1998.
- **TAG links to various MPI web-based references**
- **Using MPI on NCCS computers**
- **Good reference + tutorials.**
- **Heterogeneous Computing with MPI**

Please forward your feedback, questions, and suggestions to xrtag@nccs, or call the TAG help desk number: (301) 286-9120.

Privacy/Security Warning

Author: *NCCS Technical Assistance Group (TAG)*
Authorizing Technical Official: *W. Phillip Webster, Code 931, GSFC/NASA*
Authorizing NASA Official: *Nancy Palm, Branch Head, Code 931, GSFC/NASA*
Last Updated:*03/07/01*
Reason for Change: *New*