Welcome ...

... to the introductory workshop in MPI programming at UNICC

Schedule:
08.00-12.00 Hard work and a short coffee break

Scope of the workshop:
We will go through the basics of MPI-programming and run some simple MPI-programs (Fortran 77) on the UNICC machines.

URLs:
Workshop homepage:
http://www.tfd.chalmers.se/~hani/mpi/mpi-workshop.html
UNICC official homepage:
http://www.cs.chalmers.se/Support/UNICC/
MPI-support homepage:
http://www.tfd.chalmers.se/~hani/mpi/mpi-support.html

Outline

- Message Passing Fundamentals
- Getting Started with MPI
- Point-to-Point Communication
- Collective Communications
- Portability issues
- Further issues

Outline

- Message Passing Fundamentals
- Getting Started with MPI
- Point-to-Point Communication
- Collective Communications
- Portability issues
- Further issues

Message Passing Fundamentals

MPI is a set of functions (in C) or subroutines (in Fortran) that can be called from your code to handle communication between different processes that belong to the same task.

MPI can be used both to distribute CPU work load on several CPUs, and to distribute the memory requirement for tasks that cannot be fitted into the memory of a single CPU. In both cases the result is that the code runs faster, ideally scaling with the number of CPUs used.

The resources at UNICC are designed for this kind of parallelization and the aim of this MPI-support is to increase the usage of MPI at UNICC.

Parallel Architectures

Parallel computers have two basic architectures:

- **Distributed memory**: Each processor has its own local memory and access to the memory of other nodes via some sort of high-speed communications network. Data are exchanged between nodes as messages over the network. Hive (1 CPU/Node)

- **Shared memory**: The processors share a global memory space via a high-speed memory bus. This global memory space allows the processors to efficiently exchange or share access to data.

- **Mix of the above**: Helios (2 CPUs/Node), Helios64 (4 CPUs/Node)

Problem decomposition

There are two kinds of problem decompositions:

- **Domain decomposition**: Data are divided into pieces of approximately the same size and then mapped to different processes. Each process then works only on the portion of the data that is assigned to it. Of course, the processes may need to communicate periodically in order to exchange data. Single-Program-Multiple-Data (SPMD) follows this model where the code is identical on all processes.

- **Functional decomposition**: The problem is decomposed into a large number of smaller tasks and then, the tasks are assigned to the processes as they become available. Processes that finish quickly are simply assigned more work. Task parallelism is implemented in a master-slave paradigm.

Directive based alternative

In a directives-based data-parallel language, such as High Performance Fortran (HPF) or OpenMP, a serial code is made parallel by adding directives (which appear as comments in the serial code) that tell the compiler how to distribute data and work across the processes. The details of how data distribution, computation, and communications are to be done are left to the compiler.

Data parallel languages are usually implemented on shared memory architectures because the global memory space greatly simplifies the writing of compilers.

In the message passing approach (MPI), it is left up to the programmer to explicitly divide data and work across the processes as well as manage the communications among them. This approach is very flexible.

Parallel Programming Issues

In order to get good parallel performance:

- **Load balancing**: Equal work / CPU-time on each processor.
- **Minimizing communication**: Packing, sending, receiving, unpacking and waiting for messages takes time. Send few messages with much information rather than many messages with little information.
- **Overlapping communication and computation**: Non-blocking communication allows work to be done while waiting for messages. May be difficult in practice.

Course Problem

Description:
The initial problem is a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise:
Write a description of a parallel approach to solving the above problem.

Course Problem, a solution

A master- and slave and domain decomposition solution, where the master is responsible for I/O and communication and the slaves search different sections of the array.

Tasks for the master process:

- Read in the target and the entire integer array from the input file.
- Send the target to each of the slave processes.
- Send different sections of the array to each of the slave processes. Here, the domain (the entire integer array) is broken into smaller parts that each slave process will work on in parallel.
- Receive from the slave process target locations (as they find them).
- Write the target locations to the output file (as the master gets them).

Tasks for the slave processes:

- Receive from the master the value for the target.
- Receive from the master the subarray it is supposed to search.
- Completely scan through its subarray, sending the locations of the target to the master as it finds them.

Outline

The message passing model

- A parallel computation consists of a number of *processes* with local data and local variables.
- The *processes* are distributed on one or several *processors*
- Data is shared between processes by sending and receiving.
- Can be used on a wide variety of platforms, from shared memory multiprocessors to heterogeneous networks of workstations
- Allows more control and can achieve higher performance than the shared memory model

What is MPI?

- MPI (Message Passing Interface) is a *standard* for message passing that the vendor of a platform must follow, thus ensuring portability
- A library of functions (in C) or subroutines (in Fortran)
- MPI-1 specifies names, calling sequences and results of subroutines and functions
- MPI-2 provides additional features such as parallel I/O, C++ and Fortran 90 bindings, and dynamic process management. Not fully available - not portable. Not part of this workshop.
- Launching an MPI-program is not part of the standard.

A first program: Hello!

(download at the workshop homepage)

```
 1     program test
 2     include ’mpif.h’
 3     integer ierr,id,lpname
 4     character*255 pname

 6     call MPI_INIT( ierr )
 7     call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierr)
 8     call MPI_GET_PROCESSOR_NAME(pname,lpname,ierr)
 9     write(6,*)’Process’,id,’ running on ’,pname(1:lpname)
10     call MPI_FINALIZE(ierr)

12     end
```

Line 2, 6 and 10 must occur in all MPI-programs.
Line 1: Program name, standard Fortran
Line 2: MPI header files contain the prototypes for MPI functions/subroutines, as well as definitions of macros, special constants, and datatypes used by MPI. An appropriate ”include” statement must appear in any source file that contains MPI function calls or constants.

A first program: Hello!

(download at the workshop homepage)

```
 1    program test
 2    include 'mpif.h'
 3    integer ierr,id,lpname
 4    character*255 pname

 6    call MPI_INIT( ierr )
 7    call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierr)
 8    call MPI_GET_PROCESSOR_NAME(pname,lpname,ierr)
 9    write(6,*)'Process',id,' running on ',pname(1:lpname)
10    call MPI_FINALIZE(ierr)

12    end
```

Line 3: Definition of integers
Line 4: Definition of characters

A first program: Hello!

(download at the workshop homepage)

```
 1      program test
 2      include 'mpif.h'
 3      integer ierr,id,lpname
 4      character*255 pname

 6      call MPI_INIT( ierr )
 7      call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierr)
 8      call MPI_GET_PROCESSOR_NAME(pname,lpname,ierr)
 9      write(6,*)'Process',id,' running on ',pname(1:lpname)
10      call MPI_FINALIZE(ierr)

12      end
```

Line 6: The first MPI routine called in any MPI program must be the initialization routine MPI_INIT. This routine establishes the MPI environment, returning an error code if there is a problem. MPI_INIT may be called only once in any program! The names of all MPI entities begin with **MPI_** to avoid conflicts. In Fortran upper case is used: MPI_INIT(ierr), MPI_COMM_WORLD ...

A first program: Hello!

(download at the workshop homepage)

```
1     program test
2     include 'mpif.h'
3     integer ierr,id,lpname
4     character*255 pname

6     call MPI_INIT( ierr )
7     call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierr)
8     call MPI_GET_PROCESSOR_NAME(pname,lpname,ierr)
9     write(6,*)'Process',id,' running on ',pname(1:lpname)
10    call MPI_FINALIZE(ierr)

12    end
```

Line 7: A process can determine its rank in a communicator (MPI_COMM_WORLD) with a call to MPI_COMM_RANK. The rank is returned in 'id'. The error code 'ierr' returned is MPI_SUCCESS if the routine ran successfully (that is, the integer returned is equal to the pre-defined integer constant MPI_SUCCESS). Thus, you can test for successful operation with *if (ierr.eq.MPI_SUCCESS) then ... end if*

Communicators

- A communicator is a handle representing a group of processes that can communicate with one another.
- The communicator name is required as an argument to all point-to-point and collective operations.
- The communicator specified in the send and receive calls must agree for communication to take place.
- Processes can communicate only if they share a communicator.
- There can be many communicators, and a given process can be a member of a number of different communicators. Within each communicator, processes are numbered consecutively (starting at 0). This identifying number is known as the rank of the process in that communicator.
- The rank is also used to specify the source and destination in send and receive calls.
- If a process belongs to more than one communicator, its rank in each can (and usually will) be different!
- MPI automatically provides a basic communicator called MPI_COMM_WORLD. It is the communicator consisting of all processes. Using MPI_COMM_WORLD, every process can communicate with every other process. You can define additional communicators consisting of subsets of the available processes.

A first program: Hello!

(download at the workshop homepage)

```
 1      program test
 2      include 'mpif.h'
 3      integer ierr,id,lpname
 4      character*255 pname

 6      call MPI_INIT( ierr )
 7      call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierr)
 8      call MPI_GET_PROCESSOR_NAME(pname,lpname,ierr)
 9      write(6,*)'Process',id,' running on ',pname(1:lpname)
10      call MPI_FINALIZE(ierr)

12      end
```

Line 8: The name of the processor on which the present process runs is returned in 'pname' and the number of characters in the name of the processor is returned in 'lpname'
Line 9: Write out information

A first program: Hello!

(download at the workshop homepage)

```
 1      program test
 2      include ’mpif.h’
 3      integer ierr,id,lpname
 4      character*255 pname

 6      call MPI_INIT( ierr )
 7      call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierr)
 8      call MPI_GET_PROCESSOR_NAME(pname,lpname,ierr)
 9      write(6,*)’Process’,id,’ running on ’,pname(1:lpname)
10      call MPI_FINALIZE(ierr)

12      end
```

Line 10: The last MPI routine called should be MPI_FINALIZE which cleans up all MPI data structures, cancels operations that never completed, etc. MPI_FINALIZE must be called by all processes; if any one process does not reach this statement, the program will appear to hang. Once MPI_FINALIZE has been called, no other MPI routines (including MPI_INIT) may be called. ’ierr’ is similar to line 6.
Line 12: End of program

A first program: Hello!

(download at the workshop homepage)

```
program test
include 'mpif.h'
integer ierr,id,lpname
character*255 pname

call MPI_INIT( ierr )
call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierr)
call MPI_GET_PROCESSOR_NAME(pname,lpname,ierr)
write(6,*)'Process',id,' running on ',pname(1:lpname)
call MPI_FINALIZE(ierr)

end
```

Each process executes the same code, including probing for its rank and processor name and printing the string. The order of the printed lines is essentially random! There is no intrinsic synchronization of operations on different processes. Each time the code is run, the order of the output lines may change.

A first program: Hello!

(download at the workshop homepage)

Typical output

```
Process 1 running on node203.unicc.chalmers.se
Process 3 running on node203.unicc.chalmers.se
Process 6 running on node214.unicc.chalmers.se
Process 5 running on node214.unicc.chalmers.se
Process 7 running on node214.unicc.chalmers.se
Process 4 running on node214.unicc.chalmers.se
Process 2 running on node203.unicc.chalmers.se
Process 0 running on node203.unicc.chalmers.se
```

Logging in, compiling and running MPI programs at Helios

- Logging in:
  ssh CID@helios.unicc.chalmers.se
  Use your CDKS password

- Get hello.f and hello_Helios from the workshop homepage using mozilla

- Make sure that MPICHPATH is set to /opt/mpich-1.2.5.2-pgi/bin in .cshrc
  (see http://www.cs.chalmers.se/Support/UNICC/Helios/mpi.html)

- mpif77 -o hello hello.f

- qsub -cwd -l s_rt=0:5:0 -pe mpich 4 hello_Helios
  (see http://www.cs.chalmers.se/Support/UNICC/Helios/kora_jobb.html#Parallella)

- SGE (Sun Grid Engine) commands: qsub, qstat -f -r, qdel <PID>
  (see http://www.cs.chalmers.se/Support/UNICC/Helios/kora_jobb.html#SGE)

Logging in, compiling and running MPI programs at Helios64

- Logging in:
  ssh CID@helios64.unicc.chalmers.se
  Use your CDKS password
- Copy files from Helios:
  cp /users/unicc_old/<CID>/hello.f .
  cp /users/unicc_old/<CID>/hello_Helios .
- Modify .cshrc:
  set COMPILER=pgi-5.1
  source /users/unicc/Common/adm/cshrc
  (see http://www.cs.chalmers.se/Support/UNICC/Helios64/mpi.html)
- mpif77 -o hello hello.f
- qsub -cwd -l s_rt=0:5:0 -pe mpich 4 hello_Helios
  (see http://www.cs.chalmers.se/Support/UNICC/Helios64/running.html)
- SGE (Sun Grid Engine) commands: qsub, qstat -f -r, qdel <PID>
  (see http://www.cs.chalmers.se/Support/UNICC/Helios64/running.html)

Logging in, compiling and running MPI programs at Hive

- Get hello.f and hello_Hive to your local computer from the workshop homepage using mozilla
- Logging in:
  ssh CID@hive.unicc.chalmers.se
  Use your CDKS password
- Copy files from local computer:
  scp CID@computer.domain.se:hello.f .
  scp CID@computer.domain.se:hello_Hive .
- mpif77 -fc=pgf77 -o hello hello.f
- qsub -l nodes=4 -l walltime=00:05:00 hello_Hive
  (see http://www.cs.chalmers.se/Support/UNICC/Hive/jobs.html)
- OpenPBS commands: qsub, qstat -f, qdel <PID>
  (see http://www.cs.chalmers.se/Support/UNICC/Hive/jobs.html)

Course Problem

Description:
The initial problem is a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise:
Write a serial version of the program.
Then write a parallel pseudo-code where MPI is correctly initialized, the processes determine and use their rank, and terminate MPI. Assume that the real code will be run on 4 processors/processes.

Course Problem, serial version

```
PROGRAM search
parameter (N=300)
integer i, target ! local variables
integer b(N) ! the entire array of integers

! File b.data has the target value
! on the first line
! The remaining 300 lines of b.data
! have the values for the b array
open(unit=10,file=”b.data”)

! File found.data will contain the
! indices of b where the target is
open(unit=11,file=”found.data”)

! Read in the target
read(10,*) target

! Read in b array
do i=1,300
    read(10,*) b(i)
end do

! Search the b array and output
! the target locations
do i=1,300
    if (b(i) .eq. target) then
        write(11,*) i
    end if
end do

END
```

Run it using search_Helios64 (modified hello_Helios64) and b.data

Course Problem, parallel pseudo-code, first part

```
PROGRAM parallel_search
INCLUDE 'mpif.h'
INTEGER rank,error

CALL MPI_INIT(error)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)

if (rank .eq. 0) then ! Master process
   read in target value from input data file
   send target value to process 1,2,3
   read in integer array b from input file
   send first third of array to process 1
   send second third of array to process 2
   send last third of array to process 3

   while (not done)
      receive target indices from slaves
      write target indices to the output file
   end while
```

Course Problem, parallel pseudo-code, second part

```
else
   receive the target from process 0
   receive my sub_array from process 0

   for each element in my subarray
     if ( element value .eq. target ) then
       convert local index into global index
       send global index to process 0
     end if
   end loop

send message to process 0 indicating my search is done

end if

CALL MPI_FINALIZE(error)
END
```

Outline

- Message Passing Fundamentals
- Getting Started with MPI
- Point-to-Point Communication
- Collective Communications
- Portability issues
- Further issues

Point-to-Point Communication

- One process (the source) sends, and another process (the destination) receives.
- In general, the source and destination processes operate asynchronously.
- Pending message has several attributes and the destination process (the receiving process) can use the attributes to determine which message to receive.
- To receive a message, a process specifies a message envelope that MPI compares to the envelopes of pending messages.
- The receiving process must be careful to provide enough storage for the entire message.

Point-to-Point Communication

Messages consist of 2 parts: the **envelope** and the **message body**.

The **envelope** of an MPI message has 4 parts:

- source - the sending process
- destination - the receiving process
- communicator - the group of processes the communication belongs to
- tag - used to classify messages

The **message body** has 3 parts:

- buffer - the message data
- datatype - the type of the message data
- count - the number of items of type datatype in buffer

Basic Datatypes - Fortran

MPI type names are used as arguments in MPI routines when a type is needed.
As a general rule, the MPI datatype given in a receive must match the MPI datatype specified in the send.

MPI Datatype	**Fortran Type**
MPI_INTEGER	integer
MPI_REAL	real
MPI_DOUBLE_PRECISION	double precision
MPI_COMPLEX	complex
MPI_CHARACTER	character(1)
MPI_LOGICAL	logical
MPI_BYTE	(none)
MPI_PACKED	(none)

In addition, MPI allows you to define arbitrary data types built from the basic types - Derived Datatypes, not part of the workshop.

Blocking Send and Receive

Blocking Send and Receive block the calling process until the communication operation is completed. Blocking creates the possibility of deadlock!
Fortran:
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)

- The input argument BUF is an array; its type should match the type given in DTYPE.
- The input arguments COUNT, DTYPE, DEST, TAG, COMM are of type INTEGER.
- The output argument IERR is of type INTEGER; it contains an error code when MPI_SEND returns.

The variables passed to MPI_SEND can be overwritten and reused immediately efter the call, although the message might not have been received yet (buffering/synchronizing).

Blocking Send and Receive

MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, STATUS, IERR)

- The output argument BUF is an array; its type should match the type in DTYPE.
- The input arguments COUNT, DTYPE, SOURCE, TAG, COMM are of type INTEGER.
- The output argument STATUS is an INTEGER array with MPI_STATUS_SIZE elements, which contains source, tag and actual count of data
- The output argument IERR is of type INTEGER; it contains an error code when MPI_RECV returns.
- The source, tag, and communicator arguments must match those of a pending message in order for the message to be received.
- Wildcard values may be used for the source (accept a message from any process) and the tag.

Have a look at simple_send_and_receive.f

Deadlock

- Deadlock occurs when 2 (or more) processes are blocked and each is waiting for the other to make progress.

- Figure out why simple_deadlock.f deadlocks, why safe_exchange.f never deadlocks, why depends_on_buffering.f might deadlock and why probable_deadlock.f is likely to deadlock!

- Delete your deadlocked jobs afterwards, using qdel <PID>

Nonblocking Send and Receive

- Nonblocking Send and Receive separate the initiation of a send or receive operation from its completion by making two separate calls to MPI. The first call initiates the operation, and the second call completes it. Between the two calls, the program is free to do other things.

- Sends and receives may be posted (initiated) by calling nonblocking routines. Posted operations are identified by request handles. Using request handles, processes can check the status of posted operations or wait for their completion.

Nonblocking Send and Receive

MPI_ISEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, REQ, IERR)

- The input argument BUF is an array; its type should match the type in DTYPE.
- The input arguments COUNT, DTYPE, DEST, TAG, COMM have type INTEGER.
- The output argument REQ has type INTEGER; it is used to identify a request handle.
- The output argument IERR has type INTEGER; it contains an error code when MPI_ISEND returns.
- **None of the arguments passed to MPI_ISEND should be read or written until the send operation is completed by another call to MPI.**

Nonblocking Send and Receive

MPI_IRECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, REQUEST, IERR)

- The output argument BUF is an array; its type should match the type in DTYPE.
- The input arguments COUNT, DTYPE, SOURCE, TAG, COMM have type INTEGER.
- The output argument REQUEST has type INTEGER; it is used to identify a request handle.
- The output argument IERR has type INTEGER; it contains an error code when MPI_IRECV returns.
- **None of the arguments passed to MPI_IRECV should be read or written until the receive operation is completed with another call to MPI.**

Nonblocking Send and Receive, completion

Completion, waiting (blocking):
MPI_WAIT(REQUEST, STATUS, IERR)

- The in/out argument REQUEST has type INTEGER.
- The output argument STATUS is an INTEGER array with MPI_STATUS_SIZE elements.
- The output argument IERR has type INTEGER and contains an error code when the call returns.
- The request argument is expected to identify a previously posted send or receive.
- If the posted operation was a receive, then the source, tag, and actual count of data received are available via the status argument.
- If the posted operation was a send, the status argument may contain an error code for the send operation (different from the error code for the call to MPI_WAIT).

Nonblocking Send and Receive, completion

Completion, testing (non-blocking):
MPI_TEST(REQUEST, FLAG, STATUS, IERR)

- The in/out argument REQUEST has type INTEGER.
- The output argument FLAG has type LOGICAL.
- The output argument STATUS is an INTEGER array with MPI_STATUS_SIZE elements.
- The output argument IERR has type INTEGER and contains an error code when the call returns.
- The request argument is expected to identify a previously posted send or receive.
- If the flag argument is true, then the posted operation is complete.
- If the flag argument is true and the posted operation was a receive, then the source, tag, and actual count of data received are available via the status argument.
- If the flag argument is true and the posted operation was a send, then the status argument may contain an error code for the send operation (not for MPI_TEST).

Nonblocking Send and Receive, Example

Investigate simple_deadlock_avoided.f, which is a non-blocking version of the blocking and deadlocking simple_deadlock.f

Send Modes

There are four send modes, standard, Synchronous (S), Ready (R) and Buffered (B), but there is only one receive mode.

Send Mode	**Blocking function**	**Nonblocking function**
Standard	MPI_SEND	MPI_ISEND
Synchronous	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND

Receive Mode	**Blocking function**	**Nonblocking function**
Only	MPI_RECV	MPI_IRECV

The blocking send functions take the same arguments (in the same order) as MPI_SEND. The nonblocking send functions take the same arguments (in the same order) as MPI_ISEND.

Send Modes

- **Standard:** Either buffered (asynchronous) or synchronized. Completed when buffered or when the receive has started. Note: the variables passed to MPI_ISEND cannot be used (should not even be read) until the send operation invoked by the call has completed. A call to MPI_TEST, MPI_WAIT or one of their variants is needed to determine completion status.

- **Synchronous:** Synchronized. Completed when the receive has started.

- **Ready:** Requires that a matching receive has already been posted at the destination process before ready mode send is called. If a matching receive has not been posted at the destination, the result is undefined. It is your responsibility to make sure the requirement is met.

- **Buffered:** Requires MPI to use buffering. The downside is that you must assume responsibility for managing the buffer. If at any point, insufficient buffer is available to complete a call, the results are undefined.

Course Problem

Description:
The initial problem is a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise:
Write the real parallel code for the search problem! Using the pseudo-code from the previous chapter as a guide, fill in all the sends and receives with calls to the actual MPI send and receive routines. For this task, use only the blocking routines. Run your parallel code using 4 processes. See if you get the same results as those obtained with the serial version. Of course, you should. See parallel_search.f for one solution.

Outline

- Message Passing Fundamentals
- Getting Started with MPI
- Point-to-Point Communication
- Collective Communications
- Portability issues
- Further issues

Collective communication

- Collective communication routines transmit data among all processes in a group.
- It is important to note that collective communication calls do not use the tag mechanism of send/receive for associating calls. Rather they are associated by order of program execution. Thus, the user must ensure that all processes execute the same collective communication calls and execute them in the same order.
- Collective communication is normally synchronized, but it is not guaranteed.

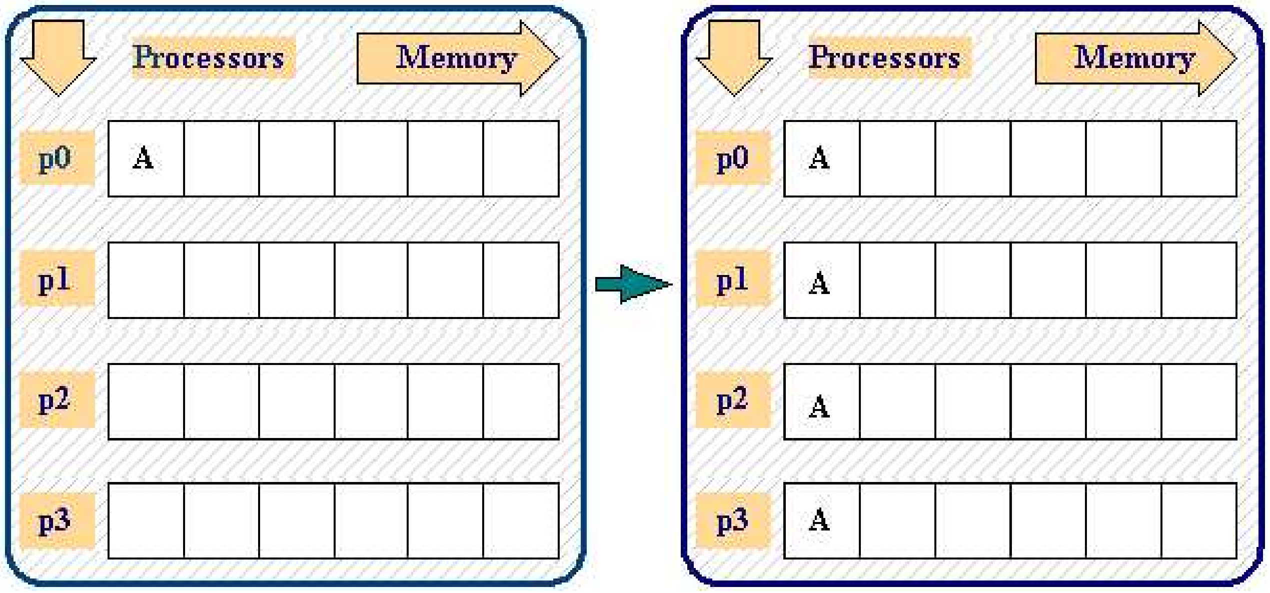
Barrier Synchronization

The MPI_BARRIER routine blocks the calling process until all group processes have called the function. When MPI_BARRIER returns, all processes are synchronized at the barrier.

```
INTEGER COMM, ERROR
MPI_BARRIER ( COMM, ERROR )
```

Broadcast

The MPI_BCAST routine enables you to copy data from the memory of the root process to the same memory locations for other processes in the communicator.

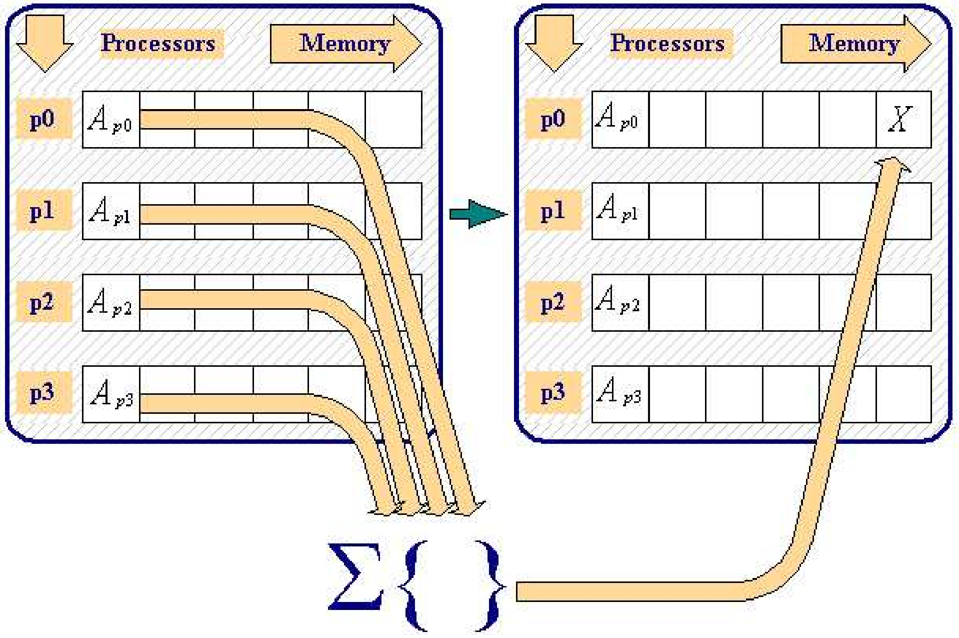


```
INTEGER COUNT, DATATYPE, ROOT, COMM, ERROR
<type> BUFFER
MPI_BCAST ( BUFFER, COUNT, DATATYPE, ROOT, COMM, ERROR )
```

See broadcast.f

Reduction

The MPI_REDUCE routine enables you to collect data from each process, reduce these data to a single value (such as a sum or max), and store the reduced result on the root process.



INTEGER COUNT, DATATYPE, OPERATION, COMM, ERROR

<datatype> SEND_BUFFER,RECV_BUFFER

MPI_Reduce (SEND_BUFFER, RECV_BUFFER, COUNT, DATATYPE, OPERATION, ROOT, COMM, ERROR)

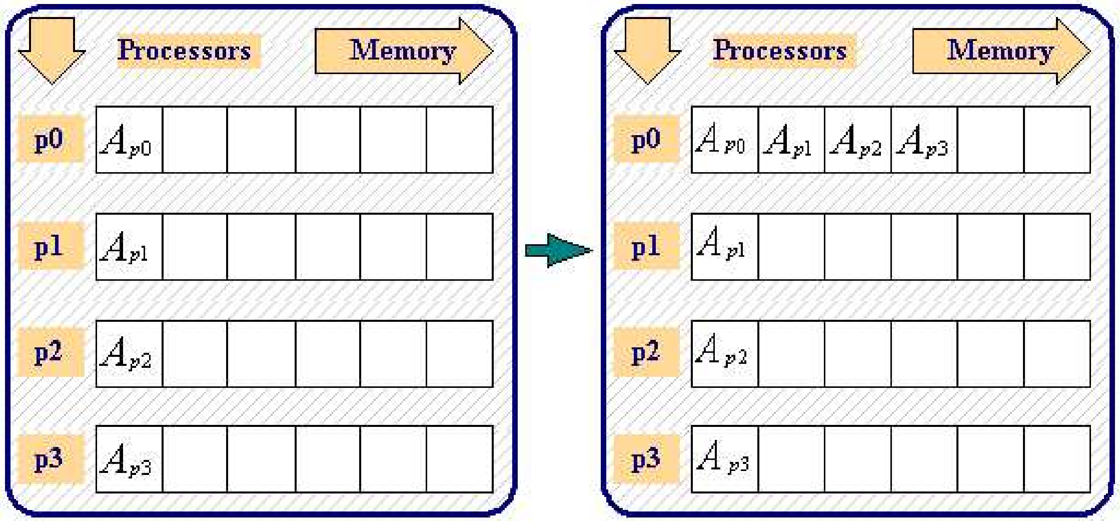
See product.f

Predefined operations available for MPI_REDUCE

Operation	**Description**
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bitwise xor
MPI_MINLOC	computes a global minimum and an index attached to the minimum value – can be used to determine the rank of the process containing the minimum value
MPI_MAXLOC	computes a global maximum and an index attached to the rank of the process containing the minimum value
user-defined	user-defined

Gather

When MPI_GATHER is called, each process (including the root process) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order.



INTEGER SEND_COUNT, SEND_TYPE, RECV_COUNT, RECV_TYPE, RANK, COMM, ERROR

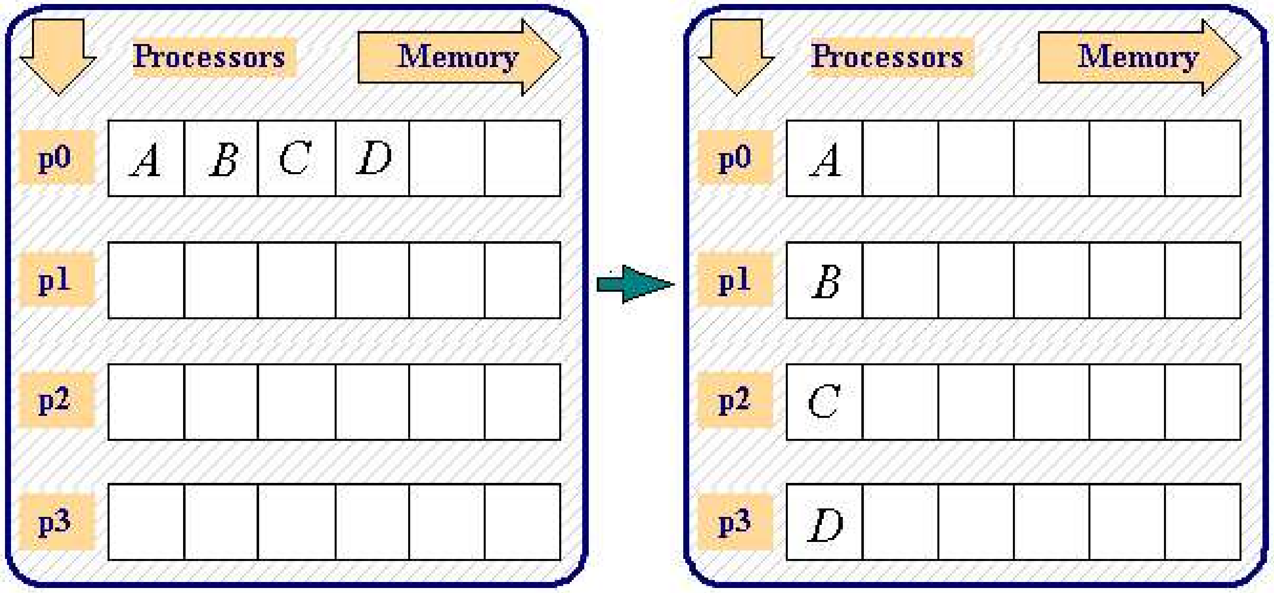
<datatype> SEND_BUFFER, RECV_BUFFER

MPI_GATHER (SEND_BUFFER, SEND_COUNT, SEND_TYPE, RECV_BUFFER, RECV_COUNT, RECV_TYPE, RANK, COMM, ERROR)

See gather.f

Scatter

When MPI_SCATTER is called, the root process breaks up a set of contiguous memory locations into equal chunks and sends one chunk to each process.



INTEGER SEND_COUNT, SEND_TYPE, RECV_COUNT, RECV_TYPE, RANK, COMM, ERROR

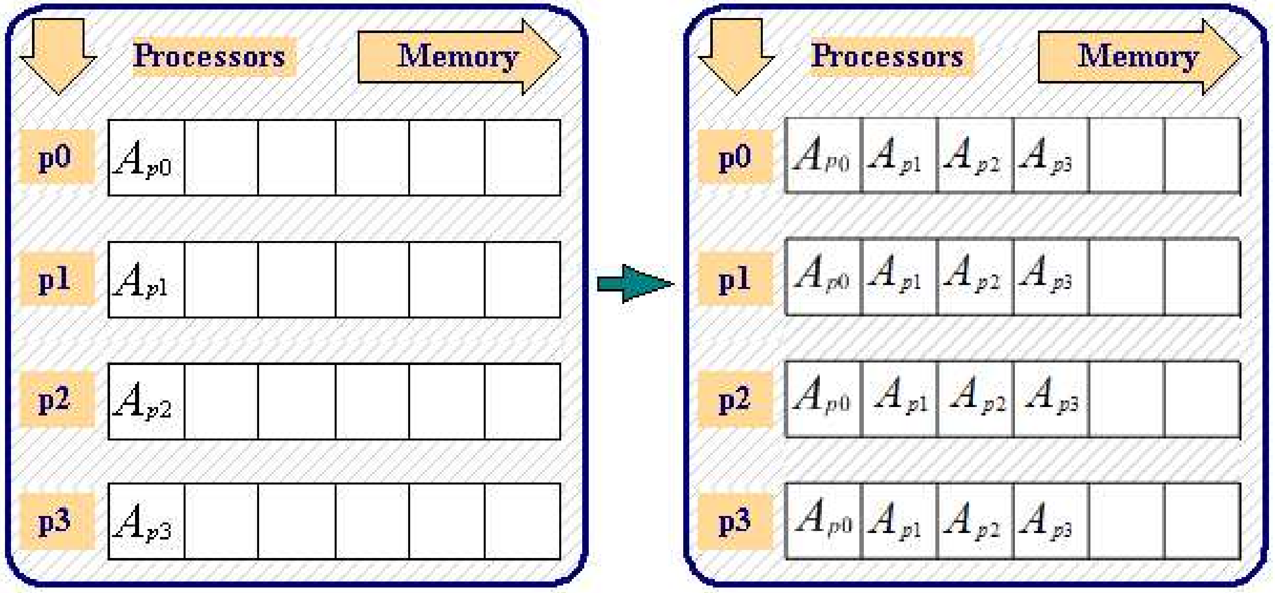
<datatype> SEND_BUFFER, RECV_BUFFER

MPI_Scatter (SEND_BUFFER, SEND_COUNT, SEND_TYPE, RECV_BUFFER, RECV_COUNT, RECV_TYPE, RANK, COMM, ERROR)

See scatter.f

Advanced Operations / MPI_ALLGATHER

After the data are gathered into the root process, you could then MPI_BCAST the gathered data to all of the other processes. It is more convenient and efficient to gather and broadcast with the single MPI_ALLGATHER operation.


Processors
Memory
p0
p1
p2
p3
A_{p0}
A_{p1}
A_{p2}
A_{p3}
Processors
Memory
p0
p1
p2
p3
A_{p0} A_{p1} A_{p2} A_{p3}
A_{p0} A_{p1} A_{p2} A_{p3}
A_{p0} A_{p1} A_{p2} A_{p3}
A_{p0} A_{p1} A_{p2} A_{p3}

The calling sequence for MPI_ALLGATHER is exactly the same as the calling sequence for MPI_GATHER.

Course Problem

Description:
The initial problem is a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise:
Modify your code parallel_search.f, to change how the master first sends out the target and subarray data to the slaves. Use the MPI broadcast routines to give each slave the target. Use the MPI scatter routine to give all processes a section of the array b it will search.
When you use the standard MPI scatter routine you will see that the global array b is now split up into four parts and the master process now has the first fourth of the array to search. So you should add a search loop (similar to the slaves') in the master section of code to search for the target and calculate the average and then write the result to the output file. This is actually an improvement in performance since all the processes perform part of the search in parallel. See parallel_search_collective.f for one solution.

Outline

Portability issues

MPI is a *standard* which is portable, but ...

- MPI is often implemented differently on different systems, which may result in a program that behaves different on different systems.
- **Buffering Assumptions.** Standard mode blocking sends and receives should not be assumed to be buffered, but synchronized.
- **Barrier Synchronization Assumptions for Collective Calls.** An MPI implementation of collective communications may or may not have the effect of barrier synchronization. One obvious exception to this is the MPI_BARRIER routine.
- **Communication Ambiguities.** When writing a program, you should make sure that messages are matched by the intended receive call. Ambiguities in the communication specification can lead to incorrect or non-deterministic programs. Use the message tags and communicators provided by MPI to avoid these types of problems.
- MPI-2 is not available on all systems.

Outline

- Message Passing Fundamentals
- Getting Started with MPI
- Point-to-Point Communication
- Collective Communications
- Portability issues
- Further issues

Further issues

- Derived Datatypes
- Communicators
- Virtual Topologies
- Parallel I/O - MPI-2
- Parallel Mathematical Libraries
- Program Performance

See the WebCT course, linked from the MPI-Support homepage.

Good luck!