Practical stuff!

**REALITY**: Ways of actually get stuff done in HPC:

- Message Passing (send, receive, broadcast, ...)  ✔ MPI
- Shared memory (load, store, lock, unlock)
- Transparent (compiler works magic)  Hahahahaha!  ✔ OpenMP
- Directive-based (compiler needs help)
- Task farming (scientific term for large transaction processing)  MapReduce/Hadoop etc
Confessions!

This presentation built out of tutorials given by:

- William Gropp (one of the founders of MPI), Argonne National Lab
- Blaise Barney, LLNL
- Rolf Rabenseifner, University of Stuttgart
- And many others ...
Message Passing Interface - MPI

- Implementation of the message passing parallel programming model
- Model for the distributed “shared nothing” environment (cheap’n’easy!)
- Performance depends on managing distributed memory usage

**Model:**
- data sent explicitly between processes
- Data distribution and data movement are programmer controlled
- **Process** is a program counter and an address space
- Same named variables in different processes are different storage (addresses)
- Processes may have multiple threads (program counters and stacks) sharing single address space
- Message passing communicates among **processes**
- Create way of moving data in one address space of distributed memory machine to another
- **Communication:** Synchronization; movement of data from one process’s address space to another; collective
- **Goal:** simplify programming but maintain performance
Message Passing Interface - MPI

- MPI has become the de facto implementation of this model
- Is a standard, a specification of a library, not a library itself
- MPI will run on any hardware (distributed, shared, hybrid memory) but clearly designed for distributed memory.
- Specification defines interface bindings for C and Fortran (C++, Java).
- Vendors/developers implement the standard
- Standard has been through a number of iterations:
  - 1980s 1990s: early parallel computing; various softwares
  - 1992: MPI meetings in Williamsburg, VI & Minneapolis, MN -> MPI Forum
  - 1994: MPI-1 Basics!
  - 1996: MPI-2 Lots of new stuff (e.g. dynamic process management, I/O, 1-sided comm)
  - 2012: MPI-3 (non-blocking collectives, sparse collectives, perf tools, deprecate C++…)
  - 80 people from 40 organisations!
- Novel features:
  - Communicators with multiple modes
  - Extensive collective operations
  - Processor topologies
  - Profiling interface
Message Passing Interface - MPI

- **Standardization is a major plus:**
  - Available on EVERY HPC platform
  - Makes your code portable
  - Vendor implementations can optimise performance for specific hardware
  - Functionality: MPI-3 defines over 4404 routines
  - Flexibility: Variety of implementations available – take your pick!

- **There are many implementations of the standard:**
  - MPICH – from Chameleon: Bill Gropp, Argonne National Lab. FREE! Foundation for many other MPI implementations (Intel MPI, IBM MPI, MVAPICH, …)
  - OpenMPI – open source (LANL, U Tennessee, U Indiana)
  - MVAPICH – Ohio State – targeted at specific hardwares


• Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.

• Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997.
Using MPI: Setup

- MPI is a library

- Need environment set up to know where MPI libraries are. Either
  - This is a one-time set up or a once per session setup (that can usually be automated)
  - Unix paths to MPI libraries etc set up correctly:
    e.g. on grape: add the following command to .cshrc
    set path = ($path /usr/mpi/gcc/openmpi-1.4.1/bin) OpenMPI
    set path = ($path /opt/mpich2-gnu/bin) MPICH
  - Can all be done with modules:
    module load <mpi implementation>
    e.g. on grape, type
    module load compilers/openmpi-1.4.1 or module load compilers/mpich

- Environment (usually) provides two other things:
  - mpif77, mpif90, mpicc – compiler wrappers that load the correct libraries
    ✓ USE AT COMPILE STAGE
  - mpirun (or mpiexec) – script that runs with the correct libraries in batch
    ✓ USE AT RUN STAGE (USUALLY IN BATCH)
Using MPI: Programming – Load headers

• MPI is a library

  => All operations are performed with subroutine calls

• When writing a program, very first thing you need to do is include the pre-made definitions of the MPI subroutine interfaces

• Basic interface definitions are in an include/header or module file:
  ➢ include ‘mpif.h’  (Fortran77 or Fortran90)
  ➢ use mpi  (Fortran90 if using modules)
  ➢ #include “mpi.h”  (C)

```fortran
program main
  include 'mpif.h'
  ...
  end

program main
  use mpi
  ...
  end
```
Next step: initialise MPI

```
program main
    include 'mpi.h'
    integer ierr
    call MPI_INIT(ierr)
    ...
    call MPI_FINALIZE(ierr)
end
```

ierr is an error code that should return ‘MPI_SUCCESS’

Most people are too lazy to check!

We need to declare it …

And then we need to quit MPI too before the end of the program
Using MPI: Programming – General bindings

### Fortran Binding

<table>
<thead>
<tr>
<th>Format:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL MPI_XXXXX(parameter, ..., ierr)</td>
</tr>
<tr>
<td>call mpi_XXXXX(parameter, ..., ierr)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL MPI_BSEND(buf, count, type, dest, tag, comm, ierr)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned as &quot;ierr&quot; parameter. MPI_SUCCESS if successful</td>
</tr>
</tbody>
</table>

### C Binding

<table>
<thead>
<tr>
<th>Format:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc = MPI_XXXXX(parameter, ... )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc = MPI_Bsend(&amp;buf, count, type, dest, tag, comm)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned as &quot;rc&quot;. MPI_SUCCESS if successful</td>
</tr>
</tbody>
</table>

- Generic
- Slight differences between C and Fortran
Using MPI: Programming – MPI comm world

- MPI uses object called communicators and groups to define which processes may communicate with each other.
- Most MPI routine require you to specify the communicator.
- Pre-defined communicator encompassing all processes available = MPI_COMM_WORLD.
- Can query this with two standard routines: MPI_COMM_SIZE, MPI_COMM_RANK.

```plaintext
program main
include 'mpi.h'
integer ierr, myid, numprocs

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

... 
call MPI_FINALIZE(ierr)
end
```

- `MPI_COMM_RANK`: returns “myid”: processor id# in comm.
- `MPI_COMM_SIZE`: returns “numprocs”: total # of procs in comm.
program hello
include 'mpi.h'
integer ierr, myid, numprocs

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

print *,"Hello world from processor ",myid," out of ",numprocs

call MPI_FINALIZE(ierr)

stop
end
Notes:

• SPMD! Every statement in the program executes independently in every processor.

• Execution could be different in each processor if we programmed it that way
  e.g. if (myid .eq. 1) then <do something> else <do something else> endif

• Output order of “hellos” from processors is undefined! Result has become non-deterministic!
Using MPI: Programming – “Hello world”

So we needed a bunch of stuff to be able to run a program!

- `ssh` to login
- Unix environment for directories etc
- `vi` to edit files
- Modules or paths for environment
- `mpif90` (or other) to compile
- Edit and submit a batch submission script
  - Knowledge of PBS/QSUB commands
  - Perhaps some scripting
  - `mpirun`
- Other batch commands to look at queue (`qstat` etc)
- Unix commands to look at output

Some grape-specific information in course web page “Resources”
program hello
include ‘mpi.h’
integer ierr, myid, numprocs

call MPI_INIT(ierr)
if (ierr .ne. MPI_SUCCESS) then
  print *,"Error starting MPI. So abort"
  call MPI_ABORT(MPI_COMM_WORLD, ierr)
endif

call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

print *,"Hello world from processor ",myid," out of ",numprocs

call MPI_FINALIZE(ierr)

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

int main(int argc, char *argv[]) {
    int numprocs, myid, rc;

    rc = MPI_Init(&argc,&argv);
    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf("Number of tasks= %d My rank= %d\n", numprocs,myid);

    /*******  do some work *******/

    MPI_Finalize();
}
```
Using MPI: Programming – Programs

From now on, the rest of the class is just what you put in this sandwich as the meat!

```fortran
program hello
  include 'mpi.h'
  integer ierr, myid, numprocs

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

  !<MEAT!!!>

  call MPI_FINALIZE(ierr)

  stop
end
```

From now on, the rest of the class is just what you put in this sandwich as the meat!
Using MPI: Programming – Programs

The “Hello world” is a (very) simple example of an *embarrassingly parallel* program.

That is, there is NO communications between processes whatsoever.

Most applications require SOME communication.

The good news: pretty much everything can be done with just two more MPI routines! So things can be done knowing just 6 routines, only two of which are non-trivial.

(Of course, maybe things can be done more easily with more routines)

The two extra routines required are

- `MPI_SEND` -- send message
- `MPI_RECV` -- receive message

There are a bunch of types of these that we will come to shortly …
MPI_SEND(address, count, datatype, dest, tag, comm, ierr)

- Message buffer is described by (address, count, datatype)
- Address is the “name” of the object to be sent.
- It is of type “datatype”. “Datatype” is a predefined MPI datatype, e.g. MPI_INT, MPI_REAL, MPI_DOUBLE (or a structure of such datatypes, or a custom designed datatype)
- We are sending “count” (integer) of the datatypes starting from the “address”
- Target process is specified by “dest” (integer) which is it’s rank in the communicator comm
- Tag is a user-defined integer, which can be arbitrary, or something actually useful (i.e. it could identify a location in an array from which the buffer was drawn)
- This is a blocking send: when the function returns, it is safe to reuse the buffer. The message may not have been delivered yet though.
<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Fortran datatype</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>
MPI_RECV(address, count, datatype, source, tag, comm, status, ierr)

• Receiving buffer is described by (address, count, datatype)
• Waits until a message is received from processor “source” in the communicator “comm” with the tag id “tag”, then places it in the buffer
• Receiving less than “count” in the message is ok, but more is an error
• Status contains more information e.g. the actual size of message received
• “source” can be MPI_ANY_SOURCE and “tag” can be MPI_ANY_TAG if do not care where message is coming from. Then use “status” to identify where it came from.

Integer status(MPI_STATUS_SIZE), recvd_from, recvd_tag, recvd_count
Call MPI_RECV(.. MPI_ANY_SOURCE, MPI_ANY_TAG, ..., status, ierr)
Recvd_tag = status(MPI_TAG)
Recvd_from = status(MPI_SOURCE)
Call MPI_GET_COUNT(status, datatype, recvd_count, ierr):wq
program simple_sendrec
include 'mpif.h'

integer myid,ierr,numprocs
integer tag,source,destination,count
integer buffer
integer stat(MPI_STATUS_SIZE)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

tag=1234
source=0
destination=1
count=1

if (myid .eq. source) then
    buffer = 5678
    call MPI_SEND(buffer, count, MPI_INTEGER, destination, tag, &
        MPI_COMM_WORLD, ierr)
    print *, 'Processor: ',myid,' sent ',buffer
endif

if (myid .eq. destination) then
    call MPI_RECV(buffer, count, MPI_INTEGER, source, tag, &
        MPI_COMM_WORLD, stat, ierr)
    print *, 'Processor: ',myid,' received ',buffer
endif

call MPI_FINALIZE(ierr)
stop
end
Using MPI:

RECAP:

6 basic MPI calls:

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

Codes written:

- Hello – embarrassingly parallel
- Sendrecv – first steps of communication
So far these have been blocking send/recvs, i.e.

- Send does not complete until the buffer is empty (available for use)
-Recv does not complete until the buffer is full (available for use)

What about a code that does the following?

**Ping pong code:**
Small extension to previous
Processor 0 sends something to processor 1 and then processor 1 sends it back

```
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>
```

What is wrong with this?
Where does a message really go?

**System buffer space is:**
- Opaque to the programmer and managed entirely by the MPI library
- A finite resource that can be easy to exhaust
- Often mysterious and not well documented!
- Able to exist on the sending side, the receiving side, or both
- Something that may improve program performance because it allows send - receive operations to be asynchronous.

**Diagram:**
- **Process 0**
  - User data
  - Local buffer
  - the network
- **Process 1**
  - Local buffer
  - User data

- **Depends on implementation!**
- **Not in programmer control**
So, this is “unsafe”!!

Possible deadlock depending on what is implementation of buffers

Imagine no buffers (send is like telephone not mail!):

send waits for receive => deadlock
Solutions to “unsafe” problem:

1. Reordering

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (1)</td>
<td>Recv (0)</td>
</tr>
<tr>
<td>Recv (1)</td>
<td>Send (0)</td>
</tr>
</tbody>
</table>

But what about if there were more than 2 such operations?

e.g. ring: everyone sends to the left? (deadlock or sequentialization!)
Solutions to “unsafe” problem:

2. Non-blocking send/receive

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>

Isend/Irecv are non-blocking

Just signals to system to send and return, or post recv and return, immediately

Do not know when they actually complete.

Must wait/synchronize at some point beyond which you need the results

(Replacings either send or recv with non-blocking works!)

Non-blocking often used to overlap communication and computation
There are “request handles” that you can test and wait on:

```fortran
integer request
integer status(MPI_STATUS_SIZE)
call MPI_Isend(start, count, datatype, 
    dest, tag, comm, request, ierr)
call MPI_Irecv(start, count, datatype, 
    dest, tag, comm, request, ierr)
call MPI_Wait(request, status, ierr)
(Each request must be waited on)
```

One can also test without waiting:
```
call MPI_Test(request, flag, status, ierr)
```

Notice:

- `MPI_Irecv` has added request and lost status in argument list
- `status` has moved to condition checker (`MPI_Wait` or `MPI+test`)
Bundled “request handles” that you can test and wait on:

```c
call MPI_Waitall(count, array_of_requests, array_of_statuses, ierr)
call MPI_Waitany(count, array_of_requests, index, status, ierr)
call MPI_Waitsome(count, array_of_requests, array_of_indices, array_of_statuses, ierr)
```

(similarly testall, testany, testsome)
Solutions to “unsafe” problem:

3. Send/recv pair:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sendrecv(1)</td>
<td>Sendrecv(0)</td>
</tr>
</tbody>
</table>

Posts a send AND a recv at SAME TIME before returning.
Solutions to “unsafe” problem: There are many solutions!

The point is, there are many versions of the point-to-point routines.

**Blocking:**

- `MPI_Recv`
- `MPI_Send`
- `MPI_Ssend` — synchronous (block until recv has started)
- `MPI_Bsend, MPI_Buffer_Attach, MPI_Buffer_Detach`
- `MPI_Rsend` -- ready send (programmer guarantees that recv has already been posted!)
- `MPI_Sendrecv` (MPI_Sendrecv_replace)
- `MPI_Wait, Waitany, Waitall, Waitsome`
- `MPI_Probe`

**Non-Blocking:**

- `MPI_IRecv`
- `MPI(ISend`  
- `MPI_ISsend`
- `MPI_IBsend, MPI_Buffer_Attach, MPI_Buffer_Detach`
- `MPI_Irsend`
- `MPI_Test, Testany, Testall, Testsome`
- `MPI_Iprobe` -- probes pending msg status for source, size, count etc
Using MPI: Programming – SEND/RECV

Synchronous mode:

Send does not complete until a matching receive has begun.

Note that “unsafe” programs deadlock

Buffered mode:

User supplies a buffer to the system for its use

User allocates enough memory to make an unsafe program safe

Ready mode:

User guarantees that a matching receive is posted

Allows access to fast protocols

Note: MPI_Recv will receive messages sent in ANY mode
Notice that you can do pretty much anything with just (some version of) send/recv

However, there are routines that make *collective* operations easier

A major chance for *OPTIMISATION*!

Collective operations involve *ALL* processors in the comm

**Collective operations:**

- **Synchronization**: barrier, …
- **Data movement**: broadcast, scatter, gather, …
- **Collective computation**: sum, …

+ In MPI-3, non-blocking collectives exist
- Only work on MPI predefined data types
Collective data movement operations:

- **MPI_BCAST**
  
  ```
  MPI_BCAST (buffer, count, datatype, root, comm, ierr)
  ```

- **MPI_SCATTER**
  
  ```
  MPI_SCATTER (sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm, ierr)
  ```

- **MPI_GATHER**
  
  ```
  MPI_GATHER (sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm, ierr)
  ```

NB. BCAST is not a multi-send!
Collective data movement operations:

- **MPI_ALLTOALL**: 
  
  ```
  MPI_ALLTOALL (sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, comm, ierr)
  ```

- **MPI_ALLGATHER**: 
  
  ```
  MPI_ALLGATHER (sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, comm, ierr)
  ```
Collective computation operations:

- **MPI_REDUCE**
  
  ```
  MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm, ierr)
  ```

- **MPI_SCAN**
  
  ```
  MPI_SCAN (sendbuf, recvbuf, count, datatype, op, comm, ierr)
  ```

( and **MPI_op_create** )
Using MPI: Programming – Collective

Scatter example:

```
program scatter
include 'mpif.h'

integer SIZE
parameter(SIZE=4)
integer numtasks, rank, sendcount, recvcount, source, ierr
real*4 sendbuf(SIZE,SIZE), recvbuf(SIZE)

! Fortran stores this array in column major order, so the
! scatter will actually scatter columns, not rows.

data sendbuf /1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0 /

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

if (numtasks .eq. SIZE) then
  source = 1
  sendcount = SIZE
  recvcount = SIZE
  call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf, &
       recvcount, MPI_REAL, source, MPI_COMM_WORLD, ierr)
  print *, 'rank=', rank, ' Results: ', recvbuf
else
  print *, 'Must specify', SIZE, ' processors. Terminating.'
endif

call MPI_FINALIZE(ierr)
end
```
Using MPI: Programming – Collective

Scatter-Gather example

```fortran
program scatter
  include 'mpif.h'

  integer SIZE
  parameter(SIZE=4)
  integer nuntasks, rank, sendcount, recvcnt, source, ierr, root
  real*4 sendbuf(SIZE*SIZE), recvbuf(SIZE)

  ! Fortran stores this array in column major order, so the
  ! scatter will actually scatter columns, not rows.
  data sendbuf /1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
    9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0 /

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nuntasks, ierr)

  if (nuntasks .eq. SIZE) then
    ! Scatter from 0
    !
    ! source = 0
    ! sendcount = SIZE
    ! recvcount = SIZE
    call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf, &
    "recvcount, MPI_REAL, source, MPI_COMM_WORLD, ierr")
    print *, 'SCATTER: rank=', rank, ' Results=', recvbuf
    ! Zero out sendbuf so can see result
    !
    sendbuf(:)=0
    !
    ! Gather to root=0
    !
    ! root = 0
    ! sendcount = SIZE
    ! recvcount = SIZE
    call MPI_GATHER(recvbuf, recvcnt, MPI_REAL, sendbuf, &
    "sendcount, MPI_REAL, root, MPI_COMM_WORLD, ierr")
    call MPI_ALLGATHER(recvbuf, recvcnt, MPI_REAL, sendbuf, &
    "sendcount, MPI_REAL, MPI_COMM_WORLD, ierr")
    !
    ! print *, 'GATHER: rank=', rank, ' Results=', sendbuf
    else
      print *, 'Must specify', SIZE, ' processors. Terminating.'
    endif

  call MPI_FINALIZE(ierr)
end
```
There are more!

- `MPI_barrier` (make all processes synchronize)
- Bcast
- `Gather`, `Gatherv`, `AllGather`, `AllGatherV`
- `Scatter`, `Scatterv`
- `AlltoAll`, `AlltoAllv`
- `Reduce`, `AllReduce`, `Reduce_Scatter`
- `Scan`

All => results delivered to all procs

V => variable size chunks

High level? Just use collectives never use point-to-point? (Data parallel approach)
e.g. Re-write the “ring” program that calculates the global sum in all processes!
Using MPI: Programming – Timing

Timing routines:

```fortran
double precision t1, t2
    t1 = MPI_Wtime()
    ...
    t2 = MPI_Wtime()
    print *, 'time is ', t2 - t1
```

Relative times (attribute MPI_WTIME_IS_GLOBAL)

MPI_Wtick – clock resolution.

Do multiple loops to time something!

Load imbalance:

```fortran
MPI_Wtime(); ... ; MPI_Barrier(MPI_COMM_WORLD); MPI_Wtime()
```
ADVANCED!

Sub-group comm

Derived data types

Virtual topologies

Dynamic process management

One-sided operations (better performance? Exposes synch? MPI_get, MPI_put)

Parallel I/O – MPI-IO, Parallel NetCDF?

Threads (cf OpenMP) hybrid prog within SMP chips
**Using MPI: Programming – Derived datatypes**

A way of combining varying data into one message

Good for data structures that need to be sent repetitively

Builds structures of basic MPI datatypes

**MPI_Type_Continuous**  
\( \text{(count, oldtype, newtype, ierr)} \)

Simplest. Just count copies of an existing datatype

**MPI_Type_vector**  
\( \text{(count, blocklen, stride, oldtype, newtype, ierr)} \)

Similar but allows for gaps (strides) in displacements

**MPI_Type_indexed**  
\( \text{(count, blocklens(), offsets(), oldtype, newtype, ierr)} \)

Array of displacements provides index to map

**MPI_Type_structure**  
\( \text{(count, blocklens(), offsets(), oldtypes(), newtype, ierr)} \)

Most general. Data map of the component datatypes

**MPI_Type_commit(datatype)**  – allocate datatype

**MPI_Type_free(datatype)**  – deallocate datatype
Using MPI: Programming – Derived datatypes

Example

Fortran derived datatype
(Keyword => store contiguously)

Declare integers

Figure out offsets: 4 reals, then 2 integers

What is offset for 4 reals?

Build structure and commit derived datatype

Actually use it in a SEND/RECV

Sample program output:

| rank= 0 | 3.00 | -3.00 | 3.00 | 0.25 | 3 | 1 |
| rank= 2 | 3.00 | -3.00 | 3.00 | 0.25 | 3 | 1 |
| rank= 1 | 3.00 | -3.00 | 3.00 | 0.25 | 3 | 1 |
| rank= 3 | 3.00 | -3.00 | 3.00 | 0.25 | 3 | 1 |
Create and use different groups of processes and their communicators:

- Group = set of processes, identified by a “handle”
- Process in a group has unique integer rank 0, …, ngroup-1
- Group has associated communicator relating the process that may communicate with each other
- Group and communicators appear similar to the programmer (communicator is a group plus a context)
- Group routines are used to specify which processes should be used to construct a communicator

Idea:

- Organize different tasks by functionality.
- Operations (e.g. collective) within sub-group

**MPI_COMM_WORLD** is the “universe” intra-communicator

- MPI-2 and above, dynamical creation of processes!
- Creation of groups is dynamic too
- Processes may be in multiple groups (but not necessarily a good idea!)

Intra- and inter- communicators …
Using MPI: Programming – Comm groups

Group Accessors

MPI_Group_size  returns number of processes in group
MPI_Group_rank   returns rank of calling process in group
MPI_Group_translate_ranks translates ranks of processes in one group to those in another group
MPI_Group_compare compares group members and group order

Group Constructors

MPI_Comm_group  returns the group associated with a communicator
MPI_Group_union  creates a group by combining two groups
MPI_Group_intersection  creates a group from the intersection of two groups
MPI_Group_difference  creates a group from the difference between two groups
MPI_Group_incl  creates a group from listed members of an existing group
MPI_Group_excl  creates a group excluding listed members of an existing group
MPI_Group_range_incl  creates a group according to first rank, stride, last rank
MPI_Group_range_excl  creates a group by deleting according to first rank, stride, last rank

Group Destructors

MPI_Group_free  marks a group for deallocation

Communicator Accessors

MPI_Comm_size  returns number of processes in communicator's group
MPI_Comm_rank   returns rank of calling process in communicator's group
MPI_Comm_compare compares two communicators

Communicator Constructors

MPI_Comm_dup  duplicates a communicator
MPI_Comm_create  creates a new communicator for a group
MPI_Comm_split  splits a communicator into multiple, non-overlapping communicators

Communicator Destructors

MPI_Comm_free  marks a communicator for deallocation
Hierarchical splitting is perhaps the best way to go.

Can split a comm by assigning processes to groups and then a newcomm to each group

Alternatively, can directly split an existing communicator (possibly easiest)

**MPI_Group_Incl(oldgroup, count, members, newgroup, ierr)**
- **oldgroup**: old group, e.g. group handle for `MPI_COMM_WORLD`. Obtain using:
  ```c
  MPI_COMM_GROUP(MPI_COMM_WORLD, oldgroup, ierr)
  ```
- **count**: no. of processes in new group
- **members**: array of size count containing ranks from oldgroup going into newgroup
- **newgroup**: new group handle

**MPI_Comm_Split(oldcomm, color, key, newcomm, ierr)**
- **oldcomm**: old comm, e.g. `MPI_COMM_WORLD`
- **color**: all processes with this color go in the newcomm
- **key**: rank of current process within newcomm (=0 => automatic determination)
- **newcomm**: resulting new communicator
Using MPI: Programming – Comm groups

```
include "mpif.h"
implicit none
integer group_world, odd_group, even_group
integer i, p, Neven, Nodd, members(0:7), ierr

call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
call MPI_Comm_group(MPI_COMM_WORLD, group_world, ierr)

Neven = (p + 1)/2 ! processes of MPI_COMM_WORLD are divided
Nodd = p - Neven ! into odd- and even-numbered groups
do i=0,Neven - 1 ! "members" determines members of even_group
  members(i) = 2*i
enddo

call MPI_Group_incl(group_world, Neven, members, even_group, ierr)
```

Call MPI_Comm_Create(MPI_COMM_WORLD, even_group, newcomm, ierr)
Notes:

Every processor involved in all groups

MPI_COMM_CREATE must be exec by all and all must pass same grp argument
Alternate (preferable?) approach:

```fortran
subroutine set_group(row_comm, col_comm)
  include 'mpif.h'
  parameter (NROW=3, NCOL=4)
  integer row_comm, col_comm, color, key

  ! Establish the row and column to which this processor belongs
  call MPI_COMM_RANK(MPI_COMM_WORLD, irank, ierr)
  irow = MOD(irank, NROW) + 1
  icol = INT(irank/NROW) + 1

  ! Build row communicators
  color = irow
  key = irank
  call MPI_COMM_SPLIT (MPI_COMM_WORLD, color, key, row_comm, ierr)

  ! Build column communicators
  color = icol
  call MPI_COMM_SPLIT (MPI_COMM_WORLD, color, key, col_comm, ierr)
  return
end
```

Notes:
Manipulation of groups is done behind the scenes
THE END
Comm/group extras
Commonly used MPI Group Routines

- `MPI_Group_size`: returns number of processes in group
- `MPI_Group_rank`: returns rank of calling process in group
- `MPI_Group_compare`: compares group members and group order
- `MPI_Group_translate_ranks`: translates ranks of processes in one group to those in another group
- `MPI_Comm_group`: returns the group associated with a communicator
- `MPI_Group_union`: creates a group by combining two groups
- `MPI_Group_intersection`: creates a group from the intersection of two groups
- `MPI_Group_difference`: creates a group from the difference between two groups
- `MPI_Group_incl`: creates a group from listed members of an existing group
- `MPI_Group_excl`: creates a group excluding listed members of an existing group
- `MPI_Group_range_incl`: creates a group according to first rank, stride, last rank
- `MPI_Group_range_excl`: creates a group by deleting according to first rank, stride, last rank
- `MPI_Group_free`: marks a group for deallocation

Commonly used MPI Communicator Routines

- `MPI_Comm_size`: returns number of processes in communicator's group
- `MPI_Comm_rank`: returns rank of calling process in communicator's group
- `MPI_Comm_compare`: compares two communicators
- `MPI_Comm_dup`: duplicates a communicator
- `MPI_Comm_create`: creates a new communicator for a group
- `MPI_Comm_split`: splits a communicator into multiple, non-overlapping communicators
- `MPI_Comm_free`: marks a communicator for deallocation

More Group and Communicator routines can be found on the MPI standard home page:

[www.mpi-forum.org](http://www.mpi-forum.org)
Using MPI: Programming – Comm groups

program group
include 'mpif.h'

integer NPROCS
parameter(NPROCS=8)
integer rank, new_rank, sendbuf, recvbuf, numtasks
integer ranks1(4), ranks2(4), ierr
integer orig_group, new_group, new_comm
data ranks1 /0, 1, 2, 3/ , ranks2 /4, 5, 6, 7/
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
if (numtasks .ne. NPROCS) then
  print *, 'Must specify NPROCS= ',NPROCS,' Terminating.'
call MPI_FINALIZE(ierr)
stop
endif
sendbuf = rank
C Extract the original group handle
call MPI_COMM_GROUP(MPI_COMM_WORLD, orig_group, ierr)
C Divide tasks into two distinct groups based upon rank
if (rank .le. NPROCS/2) then
  call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks1, &
                      new_group, ierr)
else
  call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks2, &
                      new_group, ierr)
endif

call MPI_COMM_CREATE(MPI_COMM_WORLD, new_group, &
                      new_comm, ierr)
call MPI_ALLREDUCE(sendbuf, recvbuf, 1, MPI_INTEGER, &
                      MPI_SUM, new_comm, ierr)
call MPI_GROUP_RANK(new_group, new_rank, ierr)
print *, 'rank=' ,rank,' newrank=' ,new_rank,' recvbuf=' ,recvbuf &
call MPI_FINALIZE(ierr)
end
Using MPI: Programming – Comm groups

```fortran
program Comm
  implicit none
  include "mpif.h"

  integer ierror, rank, size, rankh, sizeh, key
  integer ALL_GROUP, color, HALF_COMM
  integer N, M
  integer, dimension(MPI_STATUS_SIZE) :: status

  call MPI_Init(ierr)

  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
  call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)

  N = rank
  M = rank+10
  call MPI_COMM_GROUP(MPI_COMM_WORLD, ALL_GROUP, ierror)

  color=rank/2
  key=0

  call MPI_COMM_SPLIT(MPI_COMM_WORLD,color,key,HALF_COMM, ierror)

  call MPI_Comm_rank(HALF_COMM, rankh, ierror)
  call MPI_Comm_size(HALF_COMM, sizeh, ierror)

  write(*,*) 'Global ', rank, ' is now local rank ', rankh, ', ',color
  write(*,*) 'Global ', rank, ' local ', rankh, ' has ',N, ' and ',M

  if (rank.eq.0) then
    call MPI_SENDRECV(N,1,mpi_integer,1,1,
                      N,1,mpi_integer,1,2,HALF_COMM,status,ierror)
  else
    call MPI_SENDRECV(N,1,mpi_integer,0,2,
                      N,1,mpi_integer,0,1,HALF_COMM,status,ierror)
  endif

  write(*,*) 'Global ',rank, ' local ',rankh, ' new ',N, ' and ',M

  call MPI_Comm_free(HALF_COMM, ierror)
  call MPI_Group_free(ALL_GROUP, ierror)

  call MPI_Finalize(ierr)
end
```

Possible output of the above code (sorted afterwards)

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>3</td>
<td>local</td>
<td>1</td>
<td>has</td>
</tr>
<tr>
<td>Global</td>
<td>2</td>
<td>local</td>
<td>0</td>
<td>has</td>
</tr>
<tr>
<td>Global</td>
<td>1</td>
<td>local</td>
<td>1</td>
<td>new</td>
</tr>
<tr>
<td>Global</td>
<td>0</td>
<td>local</td>
<td>0</td>
<td>new</td>
</tr>
<tr>
<td>Global</td>
<td>2</td>
<td>local</td>
<td>0</td>
<td>new</td>
</tr>
<tr>
<td>Global</td>
<td>3</td>
<td>local</td>
<td>1</td>
<td>new</td>
</tr>
<tr>
<td>Global</td>
<td>2</td>
<td>local</td>
<td>0</td>
<td>has</td>
</tr>
<tr>
<td>Global</td>
<td>3</td>
<td>local</td>
<td>1</td>
<td>has</td>
</tr>
<tr>
<td>Global</td>
<td>0</td>
<td>local</td>
<td>0</td>
<td>new</td>
</tr>
<tr>
<td>Global</td>
<td>1</td>
<td>local</td>
<td>1</td>
<td>new</td>
</tr>
<tr>
<td>Global</td>
<td>2</td>
<td>local</td>
<td>0</td>
<td>new</td>
</tr>
<tr>
<td>Global</td>
<td>3</td>
<td>local</td>
<td>1</td>
<td>new</td>
</tr>
</tbody>
</table>
```