Intro to
MPI

http://www.oit.duke.edu/scsc
http://wiki.duke.edu/display/SCSC
scsc@duke.edu
hpc-support@duke.edu

John Pormann, Ph.D.
jbp1@duke.edu

Outline

- Overview of Parallel Computing Architectures
  - “Shared Memory” versus “Distributed Memory”

- Intro to MPI
  - Parallel programming with only 6 function calls

- Better Performance
  - Async communication (Latency Hiding)
**MPI**

MPI is short for the Message Passing Interface, an industry standard library for sending and receiving arbitrary messages into a user’s program.

- There are two major versions:
  - v.1.2 -- Basic sends and receives
  - v.2.x -- “One-sided Communication” and process-spawning

- It is fairly low-level
  - You must explicitly send data-items from one machine to another
  - You must explicitly receive data-items from other machines
  - Very easy to forget a message and have the program lock-up

- The library can be called from C/C++, Fortran (77/90/95), Java

---

**MPI in 6 Function Calls**

- **MPI_Init(…)**
  - Start-up the messaging system
  - In many cases, this also starts remote processes

- **MPI_Comm_size(…)**
  - How many parallel tasks are there?

- **MPI_Comm_rank(…)**
  - Which task number am I?

- **MPI_Send(…)**
- **MPI_Recv(…)**
- **MPI_Finalize()**
  - Shut everything down gracefully
“Hello World”

```c
#include "mpi.h"

int main( int argc, char** argv ) {
    int SelfTID, NumTasks, t, data;
    MPI_Status mpistat;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &NumTasks );
    MPI_Comm_rank( MPI_COMM_WORLD, &SelfTID );
    printf("Hello World from %i of %i\n",SelfTID,NumTasks);
    if( SelfTID == 0 ) {
        for(t=1;t<NumTasks;t++) {
            data = t;
            MPI_Send(&data,1,MPI_INT,t,55,MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(&data,1,MPI_INT,0,55,MPI_COMM_WORLD,&mpistat);
        printf("TID%i: received data=%i\n",SelfTID,data);
    }
    MPI_Finalize();
    return( 0 );
}
```

What happens in “HelloWorld”?

TID#0

MPI_Init(…)

What happens in “HelloWorld”?
What happens in “HelloWorld”? 

TID#0

MPI_Init(...) 

TID#2

MPI_Init(...) 

TID#3

MPI_Init(...) 

What happens in “HelloWorld”? 

TID#0

MPI_Comm_size(...) 
MPI_Comm_rank(...) 
printf(...) 

TID#1

MPI_Comm_size(...) 
MPI_Comm_rank(...) 
printf(...) 

TID#2

MPI_Comm_size(...) 
MPI_Comm_rank(...) 
printf(...) 

TID#3

MPI_Comm_size(...) 
MPI_Comm_rank(...) 
printf(...)
What happens in “HelloWorld”?

TID#0
if (SelfTID == ) {
  for(…)
  MPI_Send(…)
}
else {
  MPI_Recv(…)
}

TID#1
else {
  MPI_Recv(…)
}

TID#2
}
else {
  MPI_Recv(…)
}

TID#3
else {
  MPI_Recv(…)
}

Quick Note About Conventions

C vs Fortran arguments
MPI Initialization and Shutdown

- **MPI_Init( argc, argv)**
  - The odd argument list is done to allow MPI to pass the command-line arguments to all remote processes
    - Technically, only the first MPI-task gets (argc, argv) from the operating system ... it has to send them out to the other tasks
    - Don’t use (argc, argv) until AFTER MPI_Init is called

- **MPI_Finalize()**
  - Closes network connections and shuts down any other processes or threads that MPI may have created to assist with communication

Where am I? Who am I?

- **MPI_Comm_size returns the size of the “Communicator” (group of machines/tasks) that the current task is involved with**
  - MPI_COMM_WORLD means “All Machines/All Tasks”
    - You can create your own Communicators if needed, this can complicate matters as Task-5 in MPI_COMM_WORLD may be Task-0 in MyNewCommunicator

- **MPI_Comm_rank returns the “rank” of the current task inside the given Communicator ... integer ID from 0 to SIZE-1**

- These two integers are the only information you get to separate out what task should do what work within your program
MPI_Send Arguments

The MPI_Send function requires a lot of arguments to identify what data is being sent and where it is going:

\[
\text{MPI_Send(daptr, numitems, datatype, dest, tag, MPI_COMM_WORLD);}\]

- (data-pointer,num-items,data-type) specifies the data
  - MPI defines a number of data-types
    - MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_LONG_LONG
    - MPI_FLOAT, MPI_DOUBLE, MPI_COMPLEX, MPI_DOUBLE_COMPLEX
  - There are unsigned versions too ... MPI_UNSIGNED_INT
  - You can define your own data-types as well (user-defined structures)

- “Destination” identifier (integer) specifies the remote process
  - Technically, the destination ID is relative to a “Communicator” (group of machines), but MPI_COMM_WORLD is “all machines”

MPI Message Tags

All messages also must be assigned a “Message Tag”

- A programmer-defined integer value

- It is a way to give some “meaning” to the message
  - E.g. Tag=44 could mean a 10-number, integer vector, while Tag=55 is a 10-number, float vector
  - E.g. Tag=100 could be from the iterative solver while Tag=200 is used for the time-integrator

- You use it to match messages between the sender and receiver
  - A message with 3 floats which are coordinates ... Tag=99
  - A message with 3 floats which are method parameters ... Tag=88

- Extremely useful in debugging!!
  - Always use different tag numbers for every Send/Recv combination in your program
MPI_Recv Arguments

(MPI_Recv(dataptr, numitems, datatype, src, tag, MPI_COMM_WORLD, &mpistat);

- (data-pointer, num-items, data-type) specifies the data
- “source” identifier (integer) specifies the remote process
  - But can be MPI_ANY_SOURCE
- A message tag can be specified
  - Or you can use MPI_ANY_TAG

And MPI_Recv returns an “MPI_Status” object which allows you to determine:
  - Any errors that may have occurred (stat.MPI_ERROR)
  - What remote machine the message came from (stat.MPI_SOURCE)
  - What tag-number was assigned to the message (stat.MPI_TAG)
  - Can use “MPI_STATUS_IGNORE” to skip the Status object

- Obviously, this is a bit dangerous to do

Message “Matching”

(MPI requires that the (size, datatype, tag, other-task, communicator)
  match between sender and receiver
- Except for MPI_ANY_SOURCE and MPI_ANY_TAG
- Except that receive buffer can be bigger than ‘size’ sent

- Task-2 sending (100, float, 44) to Task-5
- While Task-5 waits to receive ...
  - (100, float, 44) from Task-1 -- PROBLEM
  - (100, int, 44) from Task-2 -- PROBLEM
  - (100, float, 45) from Task-2 -- PROBLEM
  - (99, float, 44) from Task-2 -- PROBLEM

  - (101, float, 44) from Task-2 -- Ok
    - But YOU need to check the recv-status and realize that array-
      item 101 in the recv-buffer is garbage
MPI_Recv with Variable Message Length

As long as your receive-buffer is BIGGER than the incoming message, you’ll be ok

```c
if( SelfTID == 0 ) {
    n = compute_message_size( ... );
    MPI_Send( send_buf, n, MPI_INT, ... );
} else {
    int recv_buf[1024];
    MPI_Recv( recv_buf, 1024, MPI_INT, ... );
}
```

You can then find out how much was actually received with:

```c
MPI_Get_count( &status, MPI_INT, &count );
```

- Returns the “count” (number of items) received

- YOU (Programmer) are responsible for allocating big-enough buffers and dealing with the actual size that was received

Error Handling

- In C, all MPI functions return an integer error code

```c
char buffer[MPI_MAX_ERROR_STRING];
int err, len;

err = MPI_Send( ... );
if( err != 0 ) {
    MPI_Error_string(err,buffer,&len);
    printf("Error %i [%s]\n",i,buffer);
}
```

- Even fancier approach:

```c
err = MPI_Send( ... );
if( err != 0 ) {
    MPI_Error_string(err,buffer,&len);
    printf("Error %i [%s] at %s:%i\n",i,buffer,__FILE__,__LINE__);
}
```
Simple Example, Re-visited

```c
#include "mpi.h"

int main(int argc, char** argv )
{
    int SelfTID, NumTasks, t, data, err;
    MPI_Status mpistat;
    err = MPI_Init( &argc, &argv );
    if( err )
    {
        printf("Error=%i in MPI_Init\n",err);
    }
    MPI_Comm_size( MPI_COMM_WORLD, &NumTasks );
    MPI_Comm_rank( MPI_COMM_WORLD, &SelfTID );
    printf("Hello World from %i of %i\n",SelfTID,NumTasks);
    if( SelfTID == 0 )
    {
        for(t=1;t<NumTasks;t++)
        {
            data = t;
            err = MPI_Send(&data,1,MPI_INT,t,100+t,MPI_COMM_WORLD);
            if( err )
            {
                printf("Error=%i in MPI_Send to %i\n",err,t);
            }
        }
    }
    else
    {
        MPI_Recv(&data,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&mpistat);
        printf("TID%i: received data=%i from src=\n",SelfTID,data, mpistat.MPI_SOURCE,mpistat.MPI_TAG);
    }
    MPI_Finalize();
    return( 0 );
}
```

Compiling MPI Programs

The MPI Standard “encourages” library-writers to provide compiler wrappers to make the build process easier:

- Usually something like ‘mpicc’ and ‘mpif77’
- But some vendors may have their own names
- This wrapper should include the necessary header-path to compile, and the necessary library(-ies) to link against, even if they are in non-standard locations
- Compiler/Linker arguments are usually passed through to the underlying compiler/linker -- so you can select ‘-O’ optimization

```bash
% mpicc -o mycode.o -c mycode.c
% mpif77 -O3 -funroll -o mysubs.o -c mysubs.f77
% mpicc -o myexe mycode.o mysubs.o
```
Running MPI Programs

- The MPI standard also “encourages” library-writers to include a program to start-up a parallel program
  - Usually called ‘mpirun’
  - But many vendors and queuing systems use something different

```bash
mpirun -np 4 myexe
```

- You explicitly define the number of tasks you want to have available to the parallel job
  - You CAN allocate more tasks than you have machines (or CPUs)
  - Each task is a separate Unix process, so if 2 or more tasks end up on the same machine, they simply time-share that machine’s resources
    - This can make debugging a little difficult since the timing of sends and recv’s will be significantly changed on a time-shared system

“Blocking” Communication

- MPI_Send and MPI_Recv are “blocking”
  - Your MPI task (program) waits until the message is sent or received before it proceeds to the next line of code
    - Note that for Sends, this only guarantees that the message has been put onto the network, not that the receiver is ready to process it
      - For large messages, it *MAY* wait for the receiver to be ready
  - Makes buffer management easier
    - When a send is complete, the buffer is ready to be re-used
  - It also means “waiting” ... bad for performance
    - What if you have other work you could be doing?
    - What if you have extra networking hardware that allows “off-loading” of the network transfer?
Blocking Communication, Example

```c
#include "mpi.h"

int main( int argc, char** argv ) {
    int SelfTID, NumTasks, t, data1, data2;
    MPI_Status mpistat;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &NumTasks );
    MPI_Comm_rank( MPI_COMM_WORLD, &SelfTID );
    printf("Hello World from %i of %i\n",SelfTID,NumTasks);
    if( SelfTID == 0 ) {
        MPI_Send(&data1,1,MPI_INT,1,55,MPI_COMM_WORLD);
        MPI_Send(&data2,1,MPI_INT,1,66,MPI_COMM_WORLD);
    } else if( SelfTID == 1 ) {
        MPI_Recv(&data2,1,MPI_INT,0,66,MPI_COMM_WORLD,&mpistat);
        MPI_Recv(&data1,1,MPI_INT,0,55,MPI_COMM_WORLD,&mpistat);
        printf("TID%i: received data=%i %i\n",SelfTID,data1,data2);
    }
    MPI_Finalize();
    return( 0 );
}
```

Any problems?

---

Blocking Communication, Example, cont’d

```
```

```
TID#0
if( SelfTID == 0 ) {
    MPI_Send( tag=55 )
    /* TID-0 blocks */
} else {
    MPI_Recv( tag=66 )
    /* TID-1 blocks */
}
```

TID#1

```
```

Messages do not match!

So both tasks could block

... and the program never completes
MPI_Isend ... “Fire and Forget” Messages

- MPI_Isend can be used to send data without the program waiting for the send to complete

  - This can make the program logic easier
    - If you need to send 10 messages, you don’t want to coordinate them with the timing of the recv’s on remote machines
  
  - But it provides less synchronization, so the programmer has to ensure that any other needed sync is provided some other way
  
  - Be careful with your buffer management -- make sure you don’t re-use the buffer until you know the data has been sent/recv’d!

  - MPI_Isend returns a “request-ID” that allows you to check if it has actually completed yet

MPI_Isend Arguments

- (data-pointer,count,data-type) as with MPI_Send
- destination, tag, communicator

- MPI_Request data-type
  - “Opaque” data-type, you can’t (shouldn’t) print it or do comparisons on it
  - It is returned by the function, and can be sent to other MPI functions which can then use the request-ID
    - MPI_Wait, MPI_Test
**MPI_Isend, cont’d**

```c
MPI_Request mpireq;
MPI_Status mpistat;
/* initial data for solver */
for(i=0;i<n;i++) {
    send_buf[i] = 0.0f;
}
MPI_Isend(send_buf, n, MPI_FLOAT, dest, tag, MPI_COMM_WORLD, &mpireq);
... 
/* do some "real" work */
... 
/* make sure last send completed */
MPI_Wait(&mpireq, &mpistat);
/* post the new data for the next iteration */
MPI_Isend(send_buf, n, MPI_FLOAT, dest, tag, MPI_COMM_WORLD, &mpireq);
```

**MPI_Irecv … “Ready to Receive”**

- **MPI_Irecv** is the non-blocking version of **MPI_Recv** (much like **Send** and **Isend**)

- Your program is indicating that it wants to eventually receive certain data, but it doesn’t need it right now
  - **MPI_Irecv** returns a “request-ID” that can be used to check if the data has been received yet
  - **NOTE:** the recv buffer is always “visible” to your program, but the data is not valid until the recv actually completes

- Very useful if many data items are to be received and you want to process the first one that arrives

- If posted early enough, an Irecv could help the MPI library avoid some extra memory-copies (i.e. better performance)
MPI_Wait and MPI_Test

To check on the progress of MPI_Isend or MPI_Irecv, you use the wait or test functions:

- MPI_Wait will block until the specified send/recv request-ID is complete
  - Returns an MPI_Status object just like MPI_Recv would
- MPI_Test will return Yes/No if the specified send/recv request-ID is complete
  - This is a non-blocking test of the non-blocking communication
  - Useful if you have lots of messages flying around and want to process the first one that arrives

Note that MPI_Isend can be paired with MPI_Recv, and MPI_Send with MPI_Irecv

- Allows different degrees of synchronization as needed by different parts of your program

```c
err = MPI_Wait( &mpireq, &mpistat );
/* message 'mpireq' is now complete */
```

```c
int flag;
err = MPI_Test( &mpireq, &flag, &mpistat );
if( flag ) {
   /* message 'mpireq' is now complete */
} else {
   /* message is still pending */
   /* do other work? */
}
```
MPI_Isend and MPI_Recv

```c
/* compute partial-sum for this task */
... */
/* launch ALL the messages at once */
for(t=0;t<NumTasks;t++) {
  if(t != SelfTID ) {
    MPI_Isend( &my_psum,1,MPI_FLOAT, t, tag,
                MPI_COMM_WORLD, &mpireq );
    MPI_Request_free( &mpireq );
  }
}
... /* do other work */
... /* we must collect all partial-sums */
  tsum = 0.0;
for(t=0;t<NumTasks;t++) {
  if(t != SelfTID ) {
    MPI_Recv( recv_psum,1,MPI_FLOAT, t, tag,
               MPI_COMM_WORLD, &mpistat );
    tsum += recv_psum;
  }
}
```

MPI_Waitany, MPI_Waitall, MPI_Testany, MPI_Testall

There are “any” and “all” versions of MPI_Wait and MPI_Test for groups of Isend/Irecv request-IDs (stored as a contiguous array)

- MPI_Waitany - waits until any one request-ID is complete, returns the array-index of the one that completed
- MPI_Testany - tests if any of the request-ID are complete, returns the array-index of the first one that it finds is complete
- MPI_Waitall - waits for all of the request-IDs to be complete
- MPI_Testall - returns Yes/No if all of the request-IDs are complete
  - This is “heavy-duty” synchronization ... if 9 out of 10 request-IDs are complete, it will still wait for the 10th one to finish
  - Be careful of your performance if you use the *all functions
**MPI_Request_free**

- The Isend/Irecv “request-IDs” are a large, though limited resource ... you can run out of them, so you need to either Wait/Test for them or Free them

```c
err = MPI_Request_free( &mpireq );
```

- Example use-case:
  - You know (based on your program’s logic) that when a certain Recv is complete, all of your previous Isends must have also completed
    - So there is no need to Wait for the Isend request-IDs
    - Make sure to use MPI_Request_free to free up the known-complete request-IDs
  - No status information is returned

---

**Why use MPI_Isend/Irecv?**

- Sometimes you just need a certain piece of data before the task can continue on to do other work … and so you just pay the cost (waiting) that comes with MPI_Recv

- But often we can find other things to do instead of just WAITING for data-item-X to be sent/received
  - Maybe we can process data-item-Y instead
  - Maybe we can do another inner iteration on a linear solver
  - Maybe we can restructure our code so that we separate our sends and receives as much as possible
    - Then we can maximize the network’s ability to ship the data around and spend less time waiting
Latency Hiding

One of the keys to avoiding Amdahl’s Law is to hide as much of the network latency (time to transfer messages) as possible.

- Compare the following …
  - Assume communication is to the TID-1/TID+1 neighbors

```cpp
def without_communication():
    MPI_Isend(... TID-1 ...);
    MPI_Isend(... TID+1 ...);
    MPI_Recv(... TID-1 ...);
    MPI_Recv(... TID+1 ...);
    x[0] += alpha*NewLeft;
    for(i=1;i<(ar_size-2);i++) {
        x[i] = alpha*(x[i-1]+x[i+1]);
    }
    x[ar_size-1] += alpha*NewRight;
```

What happens to the network in each program?

Without Latency Hiding

```cpp
def with_communication():
    MPI_Isend(... TID-1 ...);
    MPI_Isend(... TID+1 ...);
    MPI_Recv(... TID-1 ...);
    MPI_Recv(... TID+1 ...);
    x[0] += alpha*NewLeft;
    for(i=1;i<(ar_size-2);i++) {
        x[i] = alpha*(x[i-1]+x[i+1]);
    }
    x[ar_size-1] += alpha*NewRight;
```
With Latency Hiding

```c
MPI_Isend(... TID-1 ...);
MPI_Isend(... TID+1 ...);
for(i=1;i<(ar_size-2);i++) {
    x[i] = alpha*(x[i-1]+x[i+1]);
}
MPI_Recv(... TID-1 ...);
MPI_Recv(... TID+1 ...);
x[0] += alpha*NewLeft;
x[ar_size-1] += alpha*NewRight;
```

Synchronization

- The basic MPI_Recv call is “Blocking” … the calling task waits until the message is received
  - So there is already some synchronization of tasks happening

- Another major synchronization construct is a “Barrier”
  - Every task must “check-in” to the barrier before any task can leave
  - Thus every task will be at the same point in the code when all of them leave the barrier together

```c
DoWork_X(...);
MPI_Barrier( MPI_COMM_WORLD );
DoWork_Y(...);
```

- Every task will complete ‘DoWork_X()’ before any one of them starts ‘DoWork_Y()’
Barriers and Performance

- Barriers imply that, at some point in time, 9 out of 10 tasks are waiting -- sitting idle, doing nothing -- until the last task catches up
  - Clearly, “WAIT” == bad for performance!
  - You never NEED a Barrier for correctness
    - But it is a quick and easy way to make lots of race conditions and deadlocks go away!
    - MPI_Recv() implies a certain level of synchronization, maybe that is all you really need?
    - Larger barriers == more waiting
      - If Task#1 and Task#2 need to synchronize, then try to make only those two synchronize together (not all 10 tasks)
      - We’ll talk about creating custom “Communicators” later

Synchronization with MPI_Ssend

- MPI also has a “synchronous” Send call

\[
\text{MPI\_Ssend( buf, count, datatype, dest, tag, comm )}
\]

- This forces the calling task to WAIT for the receiver to post a receive operation (MPI_Recv or MPI_Irecv)
  - Only after the Recv is posted will the Sender be released to the next line of code
- The sender thus has some idea of where, in the program, the receiver currently is ... it must be at a matching recv
  - With some not-so-clever use of message-tags, you can have the MPI_Ssend operation provide fairly precise synchronization
- There is also an MPI_Issend
Collective Operations

MPI specifies a whole range of group-collective operations within MPI_Reduce and MPI_Allreduce

- Min, Max, Product, Sum, And/Or (bitwise or logical), Min-location, Max-location
- E.g. all tasks provide a partial-sum, then MPI_Reduce computes the global sum
  - And there are mechanisms to provide your own reduction operation

In theory, the MPI implementation will know how to optimize the reduction for the given network/machine architecture

Note that all reductions are barriers ... could impact performance

- You can always “do-it-yourself” with Sends/Recvs
  - And, by splitting out the Sends/Recvs, you may be able to find ways to hide some of the latency

MPI_Reduce

```c
int send_buf[4], recv_buf[4];
/* each task fills in send_buf */
for(i=0;i<4;i++) {
    send_buf[i] = ...;
}

err = MPI_Reduce( send_buf, recv_buf, 4,
                    MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD );
if( SelfTID == 0 ) {
    /* recv_buf has the min values */
    /* e.g. recv_buf[0] has the min of 
       send_buf[0] on all tasks */
} else {
    /* for all other tasks,
       recv_buf has invalid data */
}
```
MPI_Allreduce

- MPI_Reduce sends the final data to the receive buffer of the “root” task only
- MPI_Allreduce sends the final data to ALL of the receive buffers (on all of the tasks)
  - Can be useful for computing and distributing the global sum of a calculation (MPI_Allreduce with MPI_SUM)
  - ... computing and detecting convergence of a solver (MPI_MAX)
  - ... computing and detecting “eureka” events (MPI_MINLOC)

```c
err = MPI_Allreduce( send_buf, recv_buf, count,
   MPI_INT, MPI_MIN, MPI_COMM_WORLD );
```

Collective Operations, cont’d

- In general, be careful with collective operations on floats and doubles (even DIY) as they can produce different results on different processor configs
  - Running with 10 tasks splits the data into certain chunks which may accumulate round-off errors differently than the same data split onto 20 tasks
  - E.g. assume 3 digits of accuracy in the following sums across an array of size 100:

<table>
<thead>
<tr>
<th>1 Task:</th>
<th>2 Tasks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>.001</td>
<td>.001</td>
</tr>
<tr>
<td>.001</td>
<td>.001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>.001</td>
<td>.001</td>
</tr>
<tr>
<td>======</td>
<td>=====</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TID#0:</th>
<th>TID#1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>.001</td>
</tr>
<tr>
<td>.001</td>
<td>.001</td>
</tr>
<tr>
<td>.001</td>
<td>.001</td>
</tr>
<tr>
<td>======</td>
<td>=====</td>
</tr>
<tr>
<td>1.00</td>
<td>+ .050</td>
</tr>
<tr>
<td>======</td>
<td>1.05</td>
</tr>
</tbody>
</table>
Parallel Debugging

- Parallel programming has all the usual bugs that sequential programming has ... bad logic, improper arguments, array overruns

- But it also has several new kinds of bugs that can creep in
  - Deadlock
  - Race Conditions

Deadlock

- A “Deadlock” condition occurs when all Tasks are stopped at synchronization points and none of them are able to make progress
  - Sometimes this can be something simple:
    - Task-1 is waiting for a “signal” from Task-2
    - Task-2 is waiting for a “signal” from Task-1

  - Sometimes it can be more complex or cyclic:
    - Task-1 is waiting for Task-2
    - Task-2 is waiting for Task-3
    - Task-3 is waiting for ...

- A related condition is “Starvation”: when one Task needs a “lock signal” or “token” that is never sent by another Task
  - Task-i is about to send the token, but realizes it needs it again
Deadlock, cont’d

- Symptoms:
  - Program hangs (!)

- Debugging deadlock is often straightforward … unless there is also a race condition involved
  - Simple approach: put print statement in front of all 
    Recv/Barrier/Wait functions

Race Conditions

- A “Race” is a situation where multiple Tasks are making progress toward some shared or common goal where timing is critical

- Generally, this means you’ve ASSUMED something is synchronized, but you haven’t FORCED it to be synchronized

- E.g. all Tasks are trying to find a “hit” in a database
  - When a hit is detected, the Task sends a message
  - Normally, hits are rare events and so sending the message is “not a big deal”
  - But if two hits occur simultaneously, then we have a race condition
    - Task-1 and Task-2 both send a message
    - … Task-3 receives Task-1’s message first (and thinks Task-1 is the hit-owner)
    - … Task-4 receives Task-2’s message first (and thinks Task-2 is the hit-owner)
Race Conditions, cont’d

Some symptoms:

- Running the code with 4 Tasks works fine, running with 5 causes erroneous output
- Running the code with 4 Tasks works fine, running it with 4 Tasks again produces different results
- Sometimes the code works fine, sometimes it crashes after 4 hours, sometimes it crashes after 10 hours, …
- Program sometimes runs fine, sometimes hits an infinite loop
- Program output is sometimes jumbled (the usual line-3 appears before line-1)

- Any kind of indeterminacy in the running of the code, or seemingly random changes to its output, could indicate a race condition

Race Conditions, cont’d

Race Conditions can be EXTREMELY hard to debug

- Often, race conditions don’t cause crashes or even logic-errors
  - E.g. a race condition leads two Tasks to both think they are in charge of data-item-X … nothing crashes, they just keep writing and overwriting X (maybe even doing duplicate computations)
- Often, race conditions don’t cause crashes at the time they actually occur … the crash occurs much later in the execution and for a totally unrelated reason
  - E.g. a race condition leads one Task to think that the solver converged but the other Task thinks we need another iteration … crash occurs because one Task tried to compute the global residual
- Sometimes race conditions can lead to deadlock
  - Again, this often shows up as seemingly random deadlock when the same code is run on the same input
Parallel Debugging, cont’d

◆ Parallel debugging is still in its infancy
  ◆ Lots of people still use print statements to debug their code!
    ● To make matters worse, print statements alter the timing of your program ... so they can create new race conditions or alter the one you were trying to debug
  ◆ “Professional” debuggers do exist that can simplify parallel programming
    ● They try to work like a sequential debugger ... when you print a variable, it prints on all tasks; when you pause the program, it pauses on all remote machines; etc.
    ● Totalview is the biggest commercial version; Allinea is a recent contender
      ■ Sun’s Prism may still be out there

Parallel Tracing

◆ It is often useful to “see” what is going on in your parallel program
  ◆ When did a given message get sent? recv’d?
  ◆ How long did a given barrier take?

◆ You can “Trace” the MPI calls with a number of different tools
  ◆ Take time-stamps at each MPI call, then visualize the messages as arrows between different tasks

  ◆ Technically, all the “work” of MPI is done in PMPI_* functions, the MPI_* functions are wrappers
    ● So you can build your own tracing system

◆ We have the Intel Trace Visualizer installed on the DSCR
Parallel Programming Caveats

- Check return values of ALL function calls
  - REALLY check the return values of all MPI function calls

- Don’t do parallel file-write operations!! (at least not on NFS)
  - It won’t always work as you expect
  - Even flock() won’t help you in all cases
    - Push all output through a single PE

- Random numbers may not be so random, especially if you are running 1000 jobs which need 1000 “random” numbers (use SPRNG)

MPI Communicators

- A Communicator is a group of MPI-tasks
  - The Communicators must match between Send/Recv
    - ... there is no MPI_ANY_COMM flag
  - So messages CANNOT cross Communicators
    - This is useful for parallel libraries -- your library can create its own Communicator and isolate its messages from other user-code

- You create new MPI-Communicators from MPI-Groups

```c
MPI_Comm Create Comm;
MPI_Group NewGroup, OrigGroup;
int n = 4;
int array[n] = { 0,1, 4,5 }; /* TIDs to include */

MPI_Comm_group( MPI_COMM_WORLD, &OrigGroup);
MPI_Group_incl( OrigGroup, n, array, &NewGroup );
MPI_Comm_create( MPI_COMM_WORLD, NewGroup, &NewComm );
```
MPI Communicators, cont’d

Once a new Communicator is created, messages within that Communicator will have their source/destination IDs renumbered

- In the previous example, World-Task-4 could be addressed by:

```
MPI_Send(&data,1,MPI_INT,4,100,MPI_COMM_WORLD);
MPI_Send(&data,1,MPI_INT,2,100,NewComm);
```

- Using NewComm, you CANNOT send to World-Task-2

For library writers, there is a short-cut to duplicate an entire Communicator (to isolate your messages from others):

```
MPI_Comm_dups(MPI_COMM_WORLD, &NewComm);
```

- Then these messages will not conflict:

```
MPI_Send(&data,1,MPI_INT,4,100,MPI_COMM_WORLD);
MPI_Send(&data,1,MPI_INT,4,100,NewComm);
```

Where to go from here?

- [http://www.csem.duke.edu](http://www.csem.duke.edu) ... hpc-request@duke.edu

- Get on our mailing lists:
  - Scientific Visualization seminars (Friday 12-1pm)
  - SCSC seminars
  - Workshop announcements

- Follow-up seminars on specific topics
  - Parallel programming methods (OpenMP, Pthreads, MPI)
  - Performance tuning methods (Vtune, Parallel tracing)
  - Visualization tools (Amira, AVS, OpenDX)

- Contacts:
  - Dr. John Pormann, [jbp1@duke.edu](mailto:jbp1@duke.edu)
  - (Vis) Rachael Brady, [rbrady@duke.edu](mailto:rbrady@duke.edu)