Introduction to MPI and OpenMP
(with Labs)

Brandon Barker
Computational Scientist
Cornell University Center for Advanced Computing (CAC)
brandon.barker@cornell.edu

Based on materials developed by Kent Milfeld at TACC, Steve Lantz at CAC, and Brandon Barker at CAC

Workshop: High Performance Computing on Stampede 2, Jan. 23, 2017
Components

- OpenMP (*shared memory*)
  - Parallel programming on a single node
- MPI (*distributed memory*)
  - Parallel computing running on multiple nodes
- OpenMP + MPI (*hybrid computing*)
  - Combine to maximize use of HPC systems
What is OpenMP?

- OpenMP is an acronym for **Open Multi-Processing**
- An Application Programming Interface (API) for developing parallel programs in shared-memory architectures
- Three primary components of the API are:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- De facto standard -- specified for C, C++, and FORTRAN
- [http://www.openmp.org/](http://www.openmp.org/) has the specification, examples, tutorials and documentation
- OpenMP 4.5 specified November 2015
OpenMP = Multithreading

• All about executing concurrent work (tasks)
  – Tasks execute as independent *threads*
  – Threads access the same *shared memory* (no message passing!)
  – Threads synchronize only at *barriers*

• Simplest way to do multithreading – run tasks on multiple cores/units
  – Insert OpenMP *parallel directives* to create tasks for concurrent threads
  – So, shared-memory parallel programming is super-easy with OpenMP?
  – Nope! Updates to a shared variable, e.g., need special treatment…

```
// repetitive work: OK
#pragma omp parallel for
for (i=0; i<N; i++)
    a[i] = b[i] + c[i];

// repetitive updates: oops
#pragma omp parallel for
for (i=0; i<N; i++)
    sum = sum + b[i]*c[i];
```
Role of the Compiler

- OpenMP relies on the compiler to do the multithreading
  - Compiler recognizes OpenMP directives, builds in appropriate code
- A special flag is generally required to enable OpenMP
  - GNU: `gcc -fopenmp`
  - Intel: `icc -openmp`
- On the Stampede 2 login node, extra flags may be required for KNL
  - Tell the Intel compiler to use MIC-only instructions: `-xMIC-AVX512`
  - Putting it all together, e.g.: `icc -openmp -xMIC-AVX512`
  - Must do multithreading to make full use of the Xeon Phi!
OpenMP Fork-Join Parallelism

- Programs begin as a single process: master thread
- Master thread executes until a parallel region is encountered
  - Master thread creates (forks) a team of parallel threads
  - Threads in team simultaneously execute tasks in the parallel region
  - Team threads synchronize and sleep (join); master continues


**Parallel Region: C/C++**

1. `#pragma omp parallel`  
2. `{ code block`  
3. `  a = work(...);`  
4. `}`

**Line 1**  
Team of threads is formed at parallel region

**Lines 2–3**  
Each thread executes code block and subroutine call, no branching into or out of a parallel region

**Line 4**  
All threads synchronize at end of parallel region (implied barrier)

LAB: OMP Hello World  
(saw already in intro lab)
OpenMP on Shared Memory Systems

Hardware Model: Multiple Cores

Shared = accessible by all threads
x = private memory for thread x

Software Model: Threads in Parallel Region

M threads are usually mapped to M cores.
For KNL cores, 2-4 SW threads are mapped to 4 HW threads on each core.
OpenMP Directives

- OpenMP directives are comments in source code that specify parallelism for shared-memory parallel (SMP) machines.
- FORTRAN compiler directives begin with one of the sentinels `!$OMP`, `C$OMP`, or `*$OMP` – use `!$OMP` for free-format F90.
- C/C++ compiler directives begin with the sentinel `#pragma omp`.

**Fortran 90**

```fortran
!$OMP parallel
  ...
!$OMP end parallel

!$OMP parallel do
  DO ...
!$OMP end parallel do
```

**C/C++**

```c
#pragma omp parallel
  {
    ...
  }

#pragma omp parallel for
  for(...){...
```
OpenMP Syntax

- OpenMP Directives: Sentinel, construct, and clauses
  
  ```
  #pragma omp construct [clause [,]clause]...
  ```

- Example
  
  ```
  #pragma omp parallel private(i) reduction(+:sum)
  ```

- Most OpenMP constructs apply to a “structured block”, that is, a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
Worksharing Loop: C/C++

General form:

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<N; i++)
  {
    a[i] = b[i] + c[i];
  }
}
```

Line 1  Team of threads formed (parallel region).
Lines 2–6  Loop iterations are split among threads.
            Implied barrier at end of block(s) {}.

Each loop iteration must be independent of other iterations
(at a minimum, compiler will complain and your loop won’t be parallelized).
OpenMP Clauses

- **Directives** dictate what the OpenMP thread team will do
  - *Parallel regions* are marked by the `parallel` directive
  - *Worksharing loops* are marked by `do`, `for` directives (Fortran, C/C++)
- **Clauses** control the behavior of any particular OpenMP directive
  1. Scoping of variables: `private`, `shared`, `default`
  2. Initialization of variables: `copyin`, `firstprivate`
  3. Scheduling: `static`, `dynamic`, `guided`
  4. Conditional application: `if`
  5. Number of threads in team: `num_threads`
Illustration of a Race Condition

In a critical section, need *mutual exclusion* (mutex) to get intended result
- Only use when needed; incurs a performance penalty due to serial execution

The following OpenMP directives prevent this race condition:

- `#pragma omp critical` — for a code block (C/C++)
- `#pragma omp atomic` — for single statements

<table>
<thead>
<tr>
<th>Intended</th>
<th>Possible…</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>read ←</td>
<td></td>
</tr>
<tr>
<td>increment</td>
<td></td>
</tr>
<tr>
<td>write →</td>
<td></td>
</tr>
<tr>
<td>read ←</td>
<td></td>
</tr>
<tr>
<td>increment</td>
<td></td>
</tr>
<tr>
<td>write →</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ←</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>increment</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>write →</td>
<td>increment</td>
<td>1</td>
</tr>
<tr>
<td>write →</td>
<td>write →</td>
<td>1</td>
</tr>
</tbody>
</table>
OpenMP Reduction

• Recall previous example of parallel dot product
  – Simple parallel-for doesn’t work due to race condition on shared \texttt{sum}
  – Best solution is to apply OpenMP’s reduction clause
  – Doing private partial sums is fine too; add a critical section for \texttt{sum} of \texttt{ps}

```c
// repetitive reduction: OK
#pragma omp parallel for \nreduction(+:sum)
for (i=0; i<N; i++)
    sum = sum + b[i]*c[i];
```

```c
// repetitive updates: oops
int ps = 0;
#pragma omp parallel \nfirstprivate(ps)
{
    #pragma omp for
    for (i=0; i<N; i++)
        ps = ps + b[i]*c[i];
    #pragma omp critical
    sum = sum + ps;
}
```

```c
// repetitive updates: OK
#pragma omp parallel for
for (i=0; i<N; i++)
    sum = sum + b[i]*c[i];
```

### Runtime Library Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_get_num_threads()</code></td>
<td>Number of threads in current team</td>
</tr>
<tr>
<td><code>omp_get_thread_num()</code></td>
<td>Thread ID, {0: N-1}</td>
</tr>
<tr>
<td><code>omp_get_max_threads()</code></td>
<td>Number of threads in environment, <code>OMP_NUM_THREADS</code></td>
</tr>
<tr>
<td><code>omp_get_num_procs()</code></td>
<td>Number of machine CPUs</td>
</tr>
<tr>
<td><code>omp_in_parallel()</code></td>
<td>True if in parallel region &amp; multiple threads are executing</td>
</tr>
<tr>
<td><code>omp_set_num_threads(#)</code></td>
<td>Changes number of threads for parallel region, if dynamic threading is enabled</td>
</tr>
</tbody>
</table>
Environment Variables, More Functions

• To control the OpenMP runtime environment

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_NUM_THREADS</td>
<td>Set to permitted number of threads: this is the value returned by <code>omp_get_max_threads()</code></td>
</tr>
<tr>
<td>OMP_DYNAMIC</td>
<td>TRUE/FALSE for enable/disable dynamic threading by calling <code>omp_set_num_threads</code> (can also use a function to do this).</td>
</tr>
</tbody>
</table>
OpenMP 3.0 supports nested parallelism, older implementations may ignore the nesting and serialize inner parallel regions.

A nested parallel region can specify any number of threads to be used for the thread team, new id’s are assigned.
MPI: Message Passing

• Overview
• Basics
  – Hello World in MPI
  – Compiling and running MPI programs (LAB)
• MPI messages
• Point-to-point communication
  – Deadlock and how to avoid it (LAB)
• Collective communication
Overview

Introduction

• What is message passing?
  – Sending and receiving messages between tasks or processes
  – Includes performing operations on data in transit and synchronizing tasks

• Why send messages?
  – Clusters have distributed memory, i.e. each process has its own address space and no way to get at another’s

• How do you send messages?
  – Programmer makes use of an Application Programming Interface (API)
    – In this case, MPI.
  – MPI specifies the functionality of high-level communication routines
  – MPI’s functions give access to a low-level implementation that takes care of sockets, buffering, data copying, message routing, etc.
Overview

API for Distributed Memory Parallelism

• Assumption: processes do not see each other’s memory
  – Some systems overcome this assumption
    • GAS (Global Address Space) abstraction and variants
• Communication speed is determined by some kind of network
  – Typical network = switch + cables + adapters + software stack…
• Key: the implementation of MPI (or any message passing API) can be optimized for any given network
  – Expert-level performance
  – No code changes required
  – Works in shared memory, too

Image of Dell PowerEdge C8220X: http://www.theregister.co.uk/2012/09/19/dell_zeus_c8000_hyperscale_server/
Overview

Why Use MPI?

• MPI is a de facto standard for distributed memory computing
  – Public domain versions are easy to install
  – Vendor-optimized version are available on most hardware
• MPI is “tried and true”
  – MPI-1 was released in 1994, MPI-2 in 1996, and MPI-3 in 2012.
• MPI applications can be fairly portable
• MPI is a good way to learn parallel programming
• MPI is expressive: it can be used for many different models of computation, therefore can be used with many different applications
• MPI code is efficient (though some think of it as the “assembly language of parallel processing”)
• MPI has freely available implementations (e.g., MPICH, OpenMPI)
MPI and Single Program, Multiple Data (SPMD)

- One source code is written
- Same program runs multiple times, but each time with different data
- With MPI
  - Code can have conditional execution based on which processor is executing the copy: choose data
  - All copies of code are started simultaneously and may communicate and sync with each other periodically
  - Conclusion: MPI allows more SPMD programs than embarrassingly parallel applications
SPMD Programming Model

source.c ➔ a.out (compiled)

 Processor 0
    a.out
      Data 0

 Processor 1
    a.out
      Data 1

 Processor 2
    a.out
      Data 2

 Processor 3
    a.out
      Data 3
Here is the basic outline of a simple MPI program:

- Include the implementation-specific header file –
  `#include <mpi.h>` inserts basic definitions and types

- Initialize communications –
  `MPI_Init` initializes the MPI environment
  `MPI_Comm_size` returns the number of processes
  `MPI_Comm_rank` returns this process’s number (rank)

- Communicate to share data between processes –
  `MPI_Send` sends a message
  `MPI_Recv` receives a message

- Exit from the message-passing system –
  `MPI_Finalize`
Basics

Minimal Code Example: hello_mpi.c

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>

main(int argc, char **argv)
{
    char message[20];
    int i, rank, size, tag = 99;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        strcpy(message, "Hello, world!");
        for (i = 1; i < size; i++)
            MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    } else {
        MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    }
    printf("Message from process %d : %.13s\n", rank, message);
    MPI_Finalize();
}
```
Basics

Initialize and Close Environment

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>

main(int argc, char **argv)
{
    char message[20];
    int i, rank, size, tag = 99;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        strcpy(message, "Hello, world!");
        for (i = 1; i < size; i++)
            MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    } else
        MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Message from process %d : %.13s\n", rank, message);
    MPI_Finalize();
}
```

Initialize MPI environment
An implementation may also use this call as a mechanism for making the usual `argc` and `argv` command-line arguments from “main” available to all tasks (C language only).

Close MPI environment
```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

main(int argc, char **argv)
{
    char message[20];
    int i, rank, size, tag = 99;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        strcpy(message, "Hello, world!");
        for (i = 1; i < size; i++)
            MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    } else
        MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

    printf("Message from process %d : %.13s\n", rank, message);

    MPI_Finalize();
}
```

**Returns number of processes**
This, like nearly all other MPI functions, must be called after MPI_Init and before MPI_Finalize.
Input is the name of a communicator (MPI_COMM_WORLD is the global communicator) and output is the size of that communicator.

**Returns this process’ number, or rank**
Input is again the name of a communicator and the output is the rank of this process in that communicator.
Basics

Pass Messages

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>
main(int argc, char **argv)
{
    char message[20];
    int i, rank, size, tag = 99;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        strcpy(message, "Hello, world!");
        for (i = 1; i < size; i++)
            MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    } else
        MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Message from process %d : %s\n", rank, message);
    MPI_Finalize();
}
```
Basics

Compiling MPI Programs

• Generally, one uses a special compiler or wrapper script
  – Not defined by the standard
  – Consult your implementation
  – Correctly handles include path, library path, and libraries

• On Stampede 2, use MPICH-style wrappers (the most common)
  
  mpicc -o foo foo.c
  mpicxx -o foo foo.cc
  mpif90 -o foo foo.f (also mpif77)
  – Choose compiler+MPI with “module load” (default, Intel17+Intel MPI)
Running MPI Programs

• To run a simple MPI program, use MPICH-style commands
  (Can’t do this on login nodes!)
  \texttt{mpirun -n 4 ./foo} (usually mpirun is just a soft link to…)
  \texttt{mpiexec -n 4 ./foo}

• Some options for running
  \texttt{--n} -- states the number of MPI processes to launch
  \texttt{--wdir <dirname>} -- starts in the given working directory
  \texttt{--help} -- shows all options for \texttt{mpirun}

• To run over Stampede 2’s Omni-Path (as part of a batch script)
  \texttt{ibrun ./foo} (Can’t do this on login nodes either!)
  \texttt{ibrun -help ### This is OK!}
  – The scheduler handles the rest

• Note: \texttt{mpirun}, \texttt{mpiexec}, and compiler wrappers are not part of MPI,
  but they can be found in nearly all implementations
Basics  Creating an MPI Batch Script

To submit a job to the compute nodes on Stampede, you must first create a SLURM batch script with the commands you want to run.

```bash
#!/bin/bash
#SBATCH -J myMPI       # job name
#SBATCH -o myMPI.o%j   # output file (%j = jobID)
#SBATCH -e myMPI.err%j # Direct error to the error file
#SBATCH -N 1           # number of nodes requested
#SBATCH -n 16          # number of MPI (total) tasks requested
#SBATCH -p normal      # queue (partition)
#SBATCH -t 00:01:00     # run time (hh:mm:ss)
#SBATCH -A TG-TRA140011 # account number

echo 2000 > input
ibrun ./myprog < input  # run MPI executable "myprog"
```
Basics

LAB: Submitting MPI Programs

• Obtain the **hello_mpi.c** source code:

```bash
cd IntroMPI_lab/hello
```

• Compile the code using **mpicc** to output the executable **hello_mpi**

• Modify the **myMPI.sh** batch script to run **hello_mpi**
  – Do you really need the “echo” command, e.g.?
  – (see myMPI_solution.sh for corrections)

• Submit the batch script to SLURM, the batch scheduler
  – Check on progress until the job completes
  – Examine the output file

```bash
sbatch --reservation=CAC2 -p normal myMPI.sh
# see myMPI_solution.sh for hints
squeue -u <my_username>
less myMPI.o*
```
Messages

Three Parameters Describe the Data

MPI_Send(
    message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);

MPI_Recv(
    message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

Type of data, should be same for send and receive

MPI_Datatype type

Number of elements (items, not bytes)
Recv number should be greater than or equal to amount sent

int count

Address where the data start

void* data
Messages

Three Parameters Specify Routing

MPI_Send( message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD );

MPI_Recv( message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

Identify process you’re communicating with by rank number

int dest/src

Arbitrary tag number, must match up (receiver can specify MPI_ANY_TAG to indicate that any tag is acceptable)

int tag

Communicator specified for send and receive must match, no wildcards

MPI_Comm comm

Returns information on received message

MPI_Status* status
Messages

Fortran Notes

• A few Fortran particulars
  – All Fortran arguments are passed by reference
  – INTEGER ierr: variable to store the error code (in C/C++ this is the return value of the function call)

• Wildcards are allowed in C and Fortran
  – src can be the wildcard MPI_ANY_SOURCE
  – tag can be the wildcard MPI_ANY_TAG
  – status returns information on the source and tag
  – Receiver might check status when wildcards are used, e.g., to check sender rank

```fortran
mpi_send (data, count, type, dest, tag, comm, ierr)
mpi_recv (data, count, type, src, tag, comm, status, ierr)
```
Point to Point   Topics

• MPI_Send and MPI_Recv: how simple are they really?
• Blocking vs. non-blocking send and receive
• Ways to specify synchronous or asynchronous communication
• Reducing overhead: ready mode, standard mode
• Combined send/receive
• Deadlock, and how to avoid it
Point to Point  Blocking vs. Non-Blocking

**MPI_Send, MPI_Recv**

A *blocking* call suspends execution of the process until the message buffer being sent/received is safe to use.

**MPI_Isend, MPI_Irecv**

A *non-blocking* call just initiates communication; the status of data transfer and the success of the communication must be verified later by the programmer (MPI_Wait or MPI_Test).
Point to Point Send and Recv: So Many Choices

The communication mode indicates how the message should be sent.

<table>
<thead>
<tr>
<th>Communication Mode</th>
<th>Blocking Routines</th>
<th>Non-Blocking Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
</tr>
<tr>
<td>Ready</td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
</tr>
<tr>
<td>Buffered</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>Standard</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td>MPI_Irecv</td>
</tr>
<tr>
<td></td>
<td>MPI_Sendrecv</td>
<td>MPI_Sendrecv_replace</td>
</tr>
</tbody>
</table>

Note: the receive routine does not specify the communication mode -- it is simply blocking or non-blocking.
Point to Point  MPI_Sendrecv

- Good for two-way communication between a pair of nodes, in which each one sends and receives a message
- However, destination and source need not be the same (ring, e.g.)
- Equivalent to blocking send + blocking receive
- Send and receive use the same communicator but have distinct tags

\[
\text{MPI\_Sendrecv}(\text{sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status})
\]
Point to Point  Two-Way Communication: Deadlock!

• **Deadlock 1**

  IF (rank==0) THEN
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  ELSEIF (rank==1) THEN
  CALL MPI_RECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
  ENDIF

• **Deadlock 2**

  IF (rank==0) THEN
  CALL MPI_SSEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
  ELSEIF (rank==1) THEN
  CALL MPI_SSEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
  ENDIF

– MPI_Send has same problem for count*MPI_REAL > 12K
  (the MVAPICH2 “eager threshold”; it’s 256K for Intel MPI)
Basics  LAB: Deadlock

- cd to `IntroMPI_lab/deadlock`
- Compile the C or Fortran code to output the executable `deadlock`
- Create a batch script including no #SBATCH parameters:

```
cat > sr.sh
#!/bin/sh
#!/bin/sh
ibrun ./deadlock [ctrl-D to exit cat]
```

- Submit the job, specifying parameters on the command line

```
sbatch -N 1 -n 8 --reservation=CAC2 -p normal -t 00:00:30 -A TG-TRA140011 sr.sh
```

- Check job progress with `squeue`; check output with `less`.
- The program will not end normally. Edit the source code to eliminate deadlock (e.g., use `sendrecv`) and resubmit until the output is good.
Point to Point   Deadlock Solutions

• Solution 1

    IF (rank==0) THEN
    CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
    CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
    ELSEIF (rank==1) THEN
    CALL MPI_RECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
    CALL MPI_SEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
    ENDIF

• Solution 2

    IF (rank==0) THEN
    CALL MPI_SENDRECV (sendbuf,count,MPI_REAL,1,tag, &
                        recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
    ELSEIF (rank==1) THEN
    CALL MPI_SENDRECV (sendbuf,count,MPI_REAL,0,tag, &
                        recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
    ENDIF
Point to Point  More Deadlock Solutions

• Solution 3
  IF (rank==0) THEN
    CALL MPI_Irecv (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
    CALL MPI_Send (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  ELSEIF (rank==1) THEN
    CALL MPI_Irecv (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
    CALL MPI_Send (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
  ENDIF
  CALL MPI_Wait (req,status)

• Solution 4
  IF (rank==0) THEN
    CALL MPI_Bsend (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
    CALL MPI_Recv (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
  ELSEIF (rank==1) THEN
    CALL MPI_Bsend (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
    CALL MPI_Recv (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
  ENDIF
**Point to Point Two-way Communications: Summary**

<table>
<thead>
<tr>
<th></th>
<th>Task 0</th>
<th>Task 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock 1</td>
<td>Recv/Send</td>
<td>Recv/Send</td>
</tr>
<tr>
<td>Deadlock 2</td>
<td>Send/Recv</td>
<td>Send/Recv</td>
</tr>
<tr>
<td>Solution 1</td>
<td>Send/Recv</td>
<td>Recv/Send</td>
</tr>
<tr>
<td>Solution 2</td>
<td>Sendrecv</td>
<td>Sendrecv</td>
</tr>
<tr>
<td>Solution 3</td>
<td>Irecv/Send, Wait</td>
<td>(I)recv/Send, (Wait)</td>
</tr>
<tr>
<td>Solution 4</td>
<td>Bsend/Recv</td>
<td>(B)send/Recv</td>
</tr>
</tbody>
</table>
Collective Motivation

• What if one task wants to send to everyone?

```c
if (mytid == 0) {
    for (tid=1; tid<ntids; tid++) {
        MPI_Send((void*)a, /* target= */ tid, ...);
    }
} else {
    MPI_Recv((void*)a, 0, ...);
}
```

• Implements a very naive, serial broadcast

• Too primitive
  – Leaves no room for the OS / switch to optimize
  – Leaves no room for more efficient algorithms

• Too slow
Collective Overview

- Collective calls involve ALL processes within a communicator
- There are 3 basic types of collective communications:
  - Synchronization (MPI_Barrier)
  - Data movement (MPI_Bcast/Scatter/Gather/Allgather/Alltoall)
  - Collective computation/reduction (MPI_Reduce/Allreduce/Scan)
- Programming considerations & restrictions
  - Blocking operation (also non-blocking in MPI-3)
  - No use of message tag argument
  - Collective operations within subsets of processes require separate grouping and new communicator
Collective Barrier Synchronization, Broadcast

- **Barrier** blocks until all processes in comm have called it
  - Useful when measuring communication/computation time
    
    ```
    mpi_barrier(comm, ierr)
    MPI_BARRIER(comm)
    ```

- **Broadcast** sends data from root to all processes in comm
  - Again, blocks until all tasks have called it
    
    ```
    mpi_bcast(data, count, type, root, comm, ierr)
    MPI_Bcast(data, count, type, root, comm)
    ```
Collective Data Movement

- **Broadcast**
- **Scatter/Gather**
- **Allgather**
- **Alltoall**
Collective Reduction Operations

- Reduce

- Scan (Prefix)
## Collective Reduction Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bit-wise and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bit-wise or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical xor</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Logical xor</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Max value and location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Min value and location</td>
</tr>
</tbody>
</table>
Why use OpenMP and MPI?

- Try to exploit the whole shared/distributed memory hierarchy
- Memory is insufficient within one node
  - Essential concept for KNL, need to take advantage of shared data within process
  - Note: Each of Stampede 2's KNL nodes has 96GB of main memory
    - Remember, MCDRAM (16GB) is used as cache by default.
  - To run 1 process (rank) per node, have `sbatch -N x -n y`, with x == y, e.g.:
    - `#SBATCH -N 64` # number of nodes requested
    - `#SBATCH -n 64` # number of MPI (total) tasks requested
Two Views of a Stampede 2 Node

OpenMP

MPI

other nodes
#include <mpi.h>

int main(int argc, char **argv) {
    int rank, size, ie, i;
    ie = MPI_Init(&argc,&argv[]);
    ie = MPI_Comm_rank(...&rank);
    ie = MPI_Comm_size(...&size);
    //Setup shared mem, comp/comm

    //compute & communicate
    ie = MPI_Finalize();
}

Threading Example: One MPI, Many OpenMP
Some Programming Models for Intel MIC

• **OpenMP**
  – On Stampede 2, TACC expects that this is the most approachable programming model for HPC users

• **Intel Threading Building Blocks (TBB)**
  – For C++ programmers

• **Intel Cilk Plus**
  – Task-oriented add-ons for OpenMP
  – Currently for C++ programmers, may become available for Fortran

• **Intel Math Kernel Library (MKL)**
  – MKL is inherently parallelized with OpenMP

• **CAF (Actor model) library**
  – Lightweight message passing
    – (actors per thread instead of threads per task, as in Hybrid OpenMP+MPI)
  – One API for distributed and shared memory programming
  – Partially fault-tolerant (compare to MPI)
References

• Standards
  – OpenMP: http://www.openmp.org/specifications/
    • All of v4 and most of v4.5 supported by Intel Compiler 17: https://software.intel.com/en-us/node/684308
  – MPI: http://mpi-forum.org/docs/mpi-3.1/index.htm (Supported by Intel ‘17)

• CAC Virtual workshop: https://cvw.cac.cornell.edu/topics
  – Covers MPI and OpenMP in more detail
  – Corresponding Fortran examples
  – More references!
The End
Use OpenMP directives to specify worksharing in a parallel region, as well as synchronization.

```c
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    Code block
  }
}
```

**Thread action**
- **for**
  - Worksharing
- **sections**
  - Worksharing
- **single**
  - One thread
- **master**
  - One thread
- **critical**
  - One thread at a time

Parallel directives can be combined, if a parallel region has just one worksharing construct.
OpenMP Parallel Directives

- Replicated – executed by all threads
- Worksharing – divided among threads

```
PARALLEL
  {code}
END PARALLEL
```

```
PARALLEL DO
  do I = 1,N*4
    {code}
  end do
END PARALLEL DO
```

```
PARALLEL
  {code1}
DO
  do I = 1,N*4
    {code2}
  end do
  {code3}
END PARALLEL
```

Replicated

Worksharing

Combined
OpenMP Constructs

OpenMP language “extensions”

- **parallel control**: governs flow of control in the program
  - parallel directive

- **work-sharing**: distributes work among threads
  - do/for sections single directives

- **control of one task**: assigns work to a thread
  - task directive (OpenMP 3.0)

- **data access**: specifies scoping of variables
  - shared private reduction clauses

- **synchronization**: coordinates execution of threads
  - critical atomic barrier directives

- **runtime environment**: sets/gets environment
  - schedule
  - omp_set_num_threads()
  - omp_get_thread_num()
  - OMP_NUM_THREADS
  - OMP_SCHEDULE clause, API, env. variables
Private, Shared Clauses

- In the following loop, each thread needs a private copy of temp
  - The result would be unpredictable if temp were shared, because each processor would be writing and reading to/from the same location

```c
!$omp parallel do private(temp,i) shared(A,B,C)
  do i=1,N
    temp = A(i)/B(i)
    C(i) = temp + cos(temp)
  enddo
!$omp end parallel do
```

- A “lastprivate(temp)” clause will copy the last loop (stack) value of temp to the (global) temp storage when the parallel DO is complete
- A “firstprivate(temp)” initializes each thread’s temp to the global value
If work is completely parallel, scaling is linear. Scheduling, memory contention and overhead can impact speedup and Gflop/s rate.

Speedup = \frac{\text{cputime}(1)}{\text{cputime}(N)}
Overhead to Fork a Thread Team

- Increases roughly linearly with number of threads
Additional Topics to Explore...

- Schedule clause: specify how to divide work among threads
  
  ```c
  schedule(static)    schedule(dynamic,M)
  ```

- Reduction clause: perform collective operations on shared variables
  
  ```c
  reduction(+:asum)    reduction(*:aprod)
  ```

- Nowait clause: remove the barrier at the end of a parallel section
  
  ```c
  for ... nowait    end do nowait
  ```

- Lock routines: make mutual exclusion more lightweight and flexible
  
  ```c
  omp_init_lock(var)    omp_set_lock(var)
  ```

- Rectangular loop parallelization made simple
  
  ```c
  collapse(n)
  ```
Point to Point  Send and Recv: Simple?

- Sending data *from* one point (process/task)  
  *to* another point (process/task)
- One task sends while another receives
- But what if process 1 isn’t *ready* for the message from process 0?…
- MPI provides different communication modes in order to help
Point to Point  Synchronous Send, MPI_Ssend

- Handshake procedure ensures both processes are ready
- It’s likely that one of the processes will end up waiting
  - If the send call occurs first: sender waits
  - If the receive call occurs first: receiver waits
- Waiting and an extra handshake? – this could be slow
Point to Point  Buffered Send, MPI_Bsend

- Message data are copied to a system-controlled block of memory
- Process 0 continues executing other tasks without waiting
- When process 1 is ready, it fetches the message from the remote system buffer and stores it in the appropriate memory location
- Must be preceded with a call to MPI_Buffer_attach
Point to Point  Ready Send, MPI_Rsend

- Process 0 just assumes process 1 is ready! The message is sent!
- Truly simple communication, no extra handshake or copying
- But an error is generated if process 1 is unable to receive
- Only useful when logic dictates that the receiver must be ready
Point to Point  Overhead

- **System overhead**
  Buffered send has more system overhead due to the extra copy operation.

- **Synchronization overhead**
  Synchronous send has no extra copying but more waiting, because a handshake must arrive before the send can occur.

- **MPI_Send**
  Standard mode tries to trade off between the types of overhead.
  - Large messages use the “rendezvous protocol” to avoid extra copying: a handshake procedure establishes direct communication.
  - Small messages use the “eager protocol” to avoid synchronization cost: the message is quickly copied to a small system buffer on the receiver.
Point to Point  Standard Send, Eager Protocol

- Message goes a system-controlled area of memory on the receiver
- Process 0 continues executing other tasks; when process 1 is ready to receive, the system simply copies the message from the system buffer into the appropriate memory location controlled by process
- *Does not* need to be preceded with a call to MPI_Buffer_attach
Point to Point  One-Way Blocking/Non-Blocking

• Blocking send, non-blocking recv

IF (rank==0) THEN
  ! Do my work, then send to rank 1
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
  ! Do stuff that doesn't yet need recvbuf from rank 0
  CALL MPI_WAIT (req,status,ie)
  ! Do stuff with recvbuf
ENDIF

• Non-blocking send, non-blocking recv

IF (rank==0) THEN
  ! Get sendbuf ready as soon as possible
  CALL MPI_ISEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
  ! Do other stuff that doesn’t involve sendbuf
ELSEIF (rank==1) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
ENDIF
CALL MPI_WAIT (req,status,ie)
Basics  LAB: Allreduce

- cd to `IntroMPI_lab/allreduce`
- In the call to MPI_Allreduce, the reduction operation is wrong!
  - Modify the C or Fortran source to use the correct operation
- Compile the C or Fortran code to output the executable `allreduce`
- Submit the `myall.sh` batch script to SLURM, the batch scheduler
  - Check on progress until the job completes
  - Examine the output file

```bash
sbatch myall.sh
squeue -u <my_username>
less myall.o*
```

- Verify that you got the expected answer
MPI-1

- MPI-1 - Message Passing Interface (v. 1.2)
  - Library standard defined by committee of vendors, implementers, and parallel programmers
  - Used to create parallel SPMD codes based on explicit message passing
- Available on almost all parallel machines with C/C++ and Fortran bindings (and occasionally with other bindings)
- About 125 routines, total
  - 6 basic routines
  - The rest include routines of increasing generality and specificity
- This presentation has primarily covered MPI-1 routines
MPI-2

• MPI-2 includes features left out of MPI-1
  – One-sided communications
  – Dynamic process control
  – More complicated collectives
  – Parallel I/O (MPI-IO)
• Implementations of MPI-2 came along only gradually
  – Not quickly undertaken after the reference document was released (in 1997)
  – Now OpenMPI, MPICH2 (and its descendants), and the vendor implementations are nearly complete or fully complete
• Most applications still rely on MPI-1, plus maybe MPI-IO
MPI-3

- MPI-3 is largely but not strictly compatible with MPI-2
  - One-sided communication
    - Improved support for shared memory models
  - Collective communication
    - Added nonblocking functions
    - Added neighborhood collectives for specifying process topology
  - Added Fortran 2008 bindings
  - Removed C++ bindings; use C bindings from C++ instead
  - MPIT Tool Interface - allows inspection of MPI internal variables
- Not the default implementation on Stampede, but can be used, e.g:
  - module swap nvapich2/1.9a2 mvapich2-x/2.0b
  - Some implementations may not be MPI-3 complete.
MPI_COMM  MPI Communicators

• Communicators
  – Collections of processes that can communicate with each other
  – Most MPI routines require a communicator as an argument
  – Predefined communicator MPI_COMM_WORLD encompasses all tasks
  – New communicators can be defined; any number can co-exist

• Each communicator must be able to answer two questions
  – How many processes exist in this communicator?
  – MPI_Comm_size returns the answer, say, $N_p$
  – Of these processes, which process (numerical rank) am I?
  – MPI_Comm_rank returns the rank of the current process within the communicator, an integer between 0 and $N_p-1$ inclusive
  – Typically these functions are called just after MPI_Init
MPI_COMM  C Example: param.c

```c

#include <mpi.h>
main(int argc, char **argv){
    int np, mype, ierr;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &np);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &mype);
    MPI_Finalize();
}
```

#include "mpif.h"
[other includes]
int main(int argc, char *argv[]){
    int np, mype, ierr;
    [other declarations]
    :
    MPI::Init(argc, argv);
    np = MPI::COMM_WORLD.Get_size();
    mype = MPI::COMM_WORLD.Get_rank();
    :
    [actual work goes here]
    :
    MPI::Finalize();
}
program param
  include 'mpif.h'
  integer ierr, np, mype

  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD, np, ierr)
  call mpi_comm_rank(MPI_COMM_WORLD, mype, ierr)
  call mpi_finalize(ierr)
end program
## Point to Point Communication Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| **Synchronous** – sending and receiving tasks must ‘handshake’. | - Safest, therefore most portable  
- No need for extra buffer space  
- SEND/RECV order not critical | Synchronization overhead |
| **Ready** - assumes that a ‘ready to receive’ message has already been received. | - Lowest total overhead  
- No need for extra buffer space  
- Handshake not required | RECV must precede SEND |
| **Buffered** – move data to a buffer so process does not wait. | - Decouples SEND from RECV  
- No sync overhead on SEND  
- Programmer controls buffer size | Buffer copy overhead |
| **Standard** – defined by the implementer; meant to take advantage of the local system. | - Good for many cases  
- Small messages go right away  
- Large messages must sync  
- Compromise position | Your program may not be suitable |
```c
#include "mpi.h"
main(int argc, char **argv){
    int ierr, mype, myworld; double a[2];
    MPI_Status status;
    MPI_Comm icomm = MPI_COMM_WORLD;
    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(icomm, &mype);
    ierr = MPI_Comm_size(icomm, &myworld);
    if(mype == 0){
        a[0] = mype; a[1] = mype+1;
        ierr = MPI_Ssend(a,2,MPI_DOUBLE,1,9,icomm);
    }
    else if (mype == 1){
        ierr = MPI_Recv(a,2,MPI_DOUBLE,0,9,icomm,&status);
        printf("PE %d, A array= %f %f\n",mype,a[0],a[1]);
    }
    MPI_Finalize();
}
```
program oneway
    include "mpif.h"
    real*8, dimension(2) :: A
    integer, dimension(MPI_STATUS_SIZE) :: istat
    icomm = MPI_COMM_WORLD
    call mpi_init(ierr)
    call mpi_comm_rank(icomm,mype,ierr)
    call mpi_comm_size(icomm,np,ierr);
    if (mype.eq.0) then
        a(1) = dble(mype); a(2) = dble(mype+1)
        call mpi_send(A,2,MPI_REAL8,1,9,icomm,ierr)
    else if (mype.eq.1) then
        call mpi_recv(A,2,MPI_REAL8,0,9,icomm,istat,ierr)
        print '("PE",i2," received A array =",2f8.4)',mype,A
    endif
    call mpi_finalize(ierr)
end program
#include <mpi.h>
#define WCOMM MPI_COMM_WORLD
main(int argc, char **argv) {
    int npes, mype, ierr;
    double sum, val; int calc, knt=1;
    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(WCOMM, &npes);
    ierr = MPI_Comm_rank(WCOMM, &mype);

    val = (double)mype;
    ierr = MPI_Allreduce(
        &val, &sum, knt, MPI_DOUBLE, MPI_SUM, WCOMM);

    calc = (npes-1 +npes%2)*(npes/2);
    printf(" PE: %d sum=%5.0f calc=%d\n",mype,sum,calc);
    ierr = MPI_Finalize();
}
program allreduce
  include 'mpif.h'
  double precision :: val, sum
  icomm = MPI_COMM_WORLD
  knt = 1
  call mpi_init(ierr)
  call mpi_comm_rank(icomm,mype,ierr)
  call mpi_comm_size(icomm,npes,ierr)

  val = dble(mype)
  call mpi_allreduce(val,sum,knt,MPI_REAL8,MPI_SUM,icomm,ierr)

  ncalc = (npes-1 + mod(npes,2))*(npes/2)
  print '(" pe",i5," sum =",f5.0, " calc. sum =",i5)', &
             mype, sum, ncalc
  call mpi_finalize(ierr)
end program
Collective

The Collective Collection!

1/23/2017
References

• MPI standards
  – http://www.mpi-forum.org/docs/
  – Documents with marked-up changes available
  – Other mirror sites: http://www.mcs.anl.gov/mpi/
  – Freely available implementations
    • MPICH, http://www.mcs.anl.gov/mpi/mpich
    • Open MPI, http://www.open-mpi.org
• CAC Virtual workshop: https://cvw.cac.cornell.edu/topics
• Books
  – Using MPI, by Gropp, Lusk, and Skjellum
  – Parallel Programming with MPI, by Peter Pacheco
  – Using MPI-2, by Gropp, Lusk and Thakur
Heterogeneous Threading, Sequential

C/C++

```c
#pragma omp parallel
{
    #pragma omp single
    { serialWork(); }
    #pragma omp for
    for(i=0; i<N; i++) {...}
}
```

MPI process, master thread

Generate parallel region

idle threads

workshare on cpu

do serial work

wait
Heterogeneous Threading, Concurrent

MPI process, master thread

serial work; nowait

Generate parallel region

workshare on cpu

assist when done in single

wait

```c
#pragma omp parallel
{
  #pragma omp single nowait
  { serialWork(); }

  #pragma omp for schedule(dynamic)
  for(i=0; i<N; i++) {...}
}
```

C/C++