Introduction to MPI and OpenMP

University of Maryland Baltimore County
Frank Willmore
willmore@tacc.utexas.edu
April 12, 2013
General Structure of MPI Programs

Process 0 (myProgram.c)
- MPI include file
- Declarations, prototypes, etc.
- Program Begins
- Initialize MPI environment
- Do work & make message passing calls
- Terminate MPI environment
- Program Ends

Parallel code begins

Process 1 (myProgram.c)
- MPI include file
- Declarations, prototypes, etc.
- Program Begins
- Initialize MPI environment
- Do work & make message passing calls
- Terminate MPI environment
- Program Ends

Serial code

Parallel code ends

Adapted from: https://computing.llnl.gov/tutorials/mpi/
MPI Execution

• Each process runs a copy of the executable: **Single Program, Multiple Data (SPMD)**

• Each process picks the portion of the work according to its rank

• Each process works independent of the other processes, except when communicating
Arguments for MPI Routine
(buffer, data count, data type, destination)

- **Buffer**: the name of a variable (including arrays and structures) that is to be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: &var1

- **Data Count**: number of data elements to be sent or received

- **Data Type**: could be elementary data type or derived

- **Destination**: the process where a message is to be delivered
Arguments for MPI Routine (source, tag, status, request)

- **Source**: indicates the process from which the message originated

- **Tag**: non-negative integer value to uniquely identify a message

- **Status**: for a receive operation, indicates the source and tag of a message

- **Request**: a unique “request number” issued by the system that can be used to check if a particular category of operation has completed or not (more on this later)
Communicators and groups are objects that are used to define which collection of processes may communicate with each other.

- Most MPI routines require a communicator as an argument.
- `MPI_COMM_WORLD` is the predefined communicator that includes all MPI processes.
- Multiple communicators and groups can be defined.

Source: https://computing.llnl.gov/tutorials/mpi/
Content Overview

• Basic concepts related to MPI

• Environment Management MPI routines

• Compiling and running MPI programs

• Types of communication
  – Point-to-Point communication routines
  – Collective communication routines

• Examples

• Summary
Every MPI Program...

• Includes the MPI header file (`mpi.h`)
• Has a routine to initialize the MPI environment (`MPI_Init`)
• Has a routine to terminate the MPI environment (`MPI_Finalize`)

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &quot;mpi.h&quot;</code></td>
<td><code>include 'mpif.h'</code></td>
</tr>
<tr>
<td><code>MPI_Init(NULL, NULL)</code></td>
<td><code>MPI_INIT(ierr)</code></td>
</tr>
<tr>
<td><code>MPI_Init(&amp;argc, &amp;argv)</code></td>
<td><code>MPI_INIT(ierr)</code></td>
</tr>
<tr>
<td><code>MPI_Xxx(. . .);</code></td>
<td><code>CALL MPI_Xxx(. . ., ierr)</code></td>
</tr>
<tr>
<td><code>MPI_Finalize()</code></td>
<td><code>MPI_FINALIZE(ierr)</code></td>
</tr>
</tbody>
</table>

Call `mpi_xxx(. . ., ierr)`
Environment Management Routines (1)

• MPI_Init initializes the MPI execution environment, must be called before any other MPI routine is called, and is invoked only once in an MPI program.

• MPI_Finalize terminates the MPI execution environment and must be called in the last.

• MPI_Comm_size determines the number of processes (size) in a communicator (comm)
  - C: MPI_Comm_size (comm,&size)
  - Fortran: MPI_COMM_SIZE (comm,size,ierr)

Note: If comm is MPI_COMM_WORLD, then size is total number of processes in the program.
Environment Management Routines (2)

•(MPI_Comm_rank) determines the number of processes within a communicator, ranges from 0 to N-1
  - C: MPI_Comm_rank (comm,&rank)
  - Fortran: MPI_COMM_RANK (comm,rank,ierr)

•(MPI_Wtime) is a timer routine that returns elapsed wall clock time in seconds
  - C: MPI_Wtime()
  - Fortran: MPI_WTIME()
Serial Program: example1.c

```c
#include <stdio.h>

int main(){

    printf("Wonderful Class!\n");

    return(0);
}
```

Compiling:
```
login3$ icc -o example1 example1.c
```

Running:
```
login3$ ./example1
Wonderful Class!
```
Serial to Parallel: `example1.c` to `mpiExample1.c`

```c
#include <stdio.h>

#include "mpi.h" \longrightarrow Include the header file "mpi.h"

int main()
{

    printf("Wonderful Class!\n");

    return(0);
}
```
Serial to Parallel: example1.c to mpiExample1.c

```c
#include <stdio.h>

#include "mpi.h"  

int main(){

  MPI_Init(NULL, NULL);  

  printf("Wonderful Class!\n");

  return(0);
}

Notice the NULL value being passed to MPI_Init. We will come back to this later.
```
Serial to Parallel: `example1.c` to `mpiExample1.c`

```c
#include <stdio.h>

#include "mpi.h"  // Include the header file "mpi.h"

int main(){

    MPI_Init(NULL, NULL);  // Start up MPI

    printf("Wonderful Class!\n");

    MPI_Finalize(); // Shut down MPI

    return(0);
}
```
Passing **NULL to MPI_Init**

- In MPI-1.1, an implementation is allowed to require that the arguments `argc` and `argv` that are passed to `main`, be also passed to `MPI_Init`

- In MPI-2, conforming implementations are required to allow applications to pass `NULL` for both the `argc` and `argv` arguments of `main`
Content Overview

• Basic concepts related to MPI

• Environment Management MPI routines

• Compiling and running MPI programs

• Types of communication
  – Point-to-Point communication routines
  – Collective communication routines

• Examples

• Summary
Compiling mpiExample1.c

- Only MPI-2 is available (MVAPICH2 and Intel MPI)
- Intel compiler available as a default option
- The GCC compiler module is also available but Intel is recommended
- Note: only 64-bit versions of the MPI libraries have been built on Stampede
Compiling `mpiExample1.c`

- Compiling the example program

```bash
login1$ mpicc -o mpiExample1 mpiExample1.c
```

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Program</th>
<th>File Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpicc</td>
<td>C</td>
<td>.c</td>
</tr>
<tr>
<td>mpicxx</td>
<td>C++</td>
<td>Intel: .C/c/cc/cpp/cxx/c++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PGI: .C/c/cc/cpp/cxx/</td>
</tr>
<tr>
<td>mpif90</td>
<td>F77/F90</td>
<td>.f, .for, .ftn, .f90, .f95, .fpp</td>
</tr>
</tbody>
</table>
Running mpiExample1.c

- To run your application on TACC resources
  - Please consult the userguide and write a job script (myJob.sh)
    http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide

  - Submit the job to the SLURM queue
    login1$ sbatch myJob.sh

  - Remember that Stampede has 16 cores per node. Also note that tasks are same as MPI processes.
#!/bin/bash

#SBATCH -J myMPI     # Job Name
#SBATCH -o myMPI.o%j # Name of the output file
#SBATCH -n 64        # Requests 16 tasks/node, 64 cores total
#SBATCH -p normal    # Queue name normal
#SBATCH -t 01:30:00   # Run time (hh:mm:ss) - 1.5 hours
#SBATCH -A xxxxx     # Mention your account name (xxxxx)

set -x

ibrun ./mpiExample1 # Run the MPI executable

Note : ibrun is a wrapper for mpirun/mpiexec that is exclusive to TACC resources
login3$ cat myMPI.o2339942
...
TACC: Starting up job 2339942
TACC: Setting up parallel environment for MVAPICH ssh-based mpirun.
TACC: Setup complete. Running job script.
TACC: starting parallel tasks...
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
...
TACC: Shutting down parallel environment.
TACC: Cleaning up after job: 2339942
TACC: Done.
Using Communicator: mpiExample2.c

1. #include <stdio.h>
2. #include "mpi.h"

3. int main(int argc, char* argv[]){

4. int rank, size; \(\text{Extend the variable declaration section}\)

5. MPI_Init(&argc, &argv); \(\text{Note argc and argv}\)

6. MPI_Comm_rank(MPI_COMM_WORLD, &rank); \(\text{Find process rank}\)

7. MPI_Comm_size(MPI_COMM_WORLD, &size); \(\text{Find out number of processes}\)

8. printf("Hello MPI World from process %d!", rank);

9. MPI_Finalize();

10. return 0;

11. }
Output from mpiExample2.c

TACC: Setup complete. Running job script.
TACC: starting parallel tasks...
Hello MPI World from process 45!
Hello MPI World from process 7!
Hello MPI World from process 10!
Hello MPI World from process 62!
Hello MPI World from process 27!
Hello MPI World from process 22!
Hello MPI World from process 33!
......
TACC: Shutting down parallel environment.
TACC: Shutdown complete. Exiting.
TACC: Cleaning up after job: 2340827
TACC: Done
Content Overview

• Basic concepts related to MPI

• Environment Management MPI routines

• Compiling and running MPI programs

• Types of communication
  – Point-to-Point communication routines
  – Collective communication routines

• Examples

• Summary
Modes of Communication

• Point-to-Point
  – Synchronous (only send)
  – Blocking (send and receive)
  – Non-blocking (send and receive)
  – Buffered (only send)
  – Combined (send/receive together)
  – "Ready" (send)

• Collective

Content Overview

• Basic concepts related to MPI

• Environment Management MPI routines

• Compiling and running MPI programs

• **Types of communication**
  – Point-to-Point communication routines
  – Collective communication routines
  – Examples

• Summary
Point-to-Point Communication

• Involve message passing between two different MPI processes

• One process performs a send operation and the other task performs a matching receive operation

• There should be a matching receive routine for every send routine
  – If a send is not paired with a matching receive then the code will have a deadlock
Point-to-Point Communication (blocking versus non-blocking)

• Blocking:
  – A blocking receive only "returns" after the data has arrived and is ready for use by the program
  – A blocking send routine returns after it is safe to modify the application buffer for reuse

• Non-blocking:
  – Non-blocking send and receive routines will return almost immediately
  – It is unsafe to modify the application buffer until you know for a fact that the requested non-blocking operation was actually performed
Point-to-Point Communication
(blocking send)

MPI_Send(void *buf, int count, MPI_Datatype dType, int dest, int tag, MPI_Comm comm)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>Initial address of the send buffer</td>
</tr>
<tr>
<td>count</td>
<td>Number of items to send</td>
</tr>
<tr>
<td>dType</td>
<td>MPI data type of items to send</td>
</tr>
<tr>
<td>dest</td>
<td>MPI rank or task that would receive the data</td>
</tr>
<tr>
<td>tag</td>
<td>Message ID</td>
</tr>
<tr>
<td>comm</td>
<td>MPI communicator where the exchange occurs</td>
</tr>
</tbody>
</table>

Some elementary data types: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, ...
Point-to-Point Communication (blocking receive)

MPI_Recv(void *buf, int count, MPI_Datatype dType, int source, int tag, MPI_Comm comm, MPI_Status *status)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>Initial address of receive buffer</td>
</tr>
<tr>
<td>count</td>
<td>Number of items to receive</td>
</tr>
<tr>
<td>dType</td>
<td>MPI data type of items to receive</td>
</tr>
<tr>
<td>source</td>
<td>MPI rank of task sending the data</td>
</tr>
<tr>
<td>tag</td>
<td>Message ID</td>
</tr>
<tr>
<td>comm</td>
<td>MPI communicator where the exchange occurs</td>
</tr>
<tr>
<td>status</td>
<td>Returns information on the message received, indicates the source of message and tag of the message</td>
</tr>
</tbody>
</table>
Point-to-Point Communication
(blocking send and receive in code)

```c
if (rank == 0) {
    dest = 1;
    source = 1;
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
             MPI_COMM_WORLD);
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
              MPI_COMM_WORLD, &Stat);
}
else if (rank == 1) {
    dest = 0;
    source = 0;
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
              MPI_COMM_WORLD, &Stat);
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
             MPI_COMM_WORLD);
}
```
Point-to-Point Communication
(this code will deadlock)

if (rank == 0) {
    dest = 1;
    source = 1;
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
              MPI_COMM_WORLD, &Stat);
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
              MPI_COMM_WORLD);
} else if (rank == 1) {
    dest = 0;
    source = 0;
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
              MPI_COMM_WORLD, &Stat);
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
              MPI_COMM_WORLD);
}
Point-to-Point Communication (non-blocking send & receive)

<table>
<thead>
<tr>
<th>Non-blocking send</th>
<th>MPI_Isend(buffer,count,type,dest,tag,comm,request)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-blocking receive</td>
<td>MPI_Irecv(buffer,count,type,source,tag,comm,request)</td>
</tr>
</tbody>
</table>

- **MPI_Request** objects are used by non-blocking send & receive calls
  - In C, this argument is a pointer to a predefined structure named MPI_Request
  - The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation
Point-to-Point Communication (MPI_Wait)

- **MPI_Wait** is a blocking routine

  ```c
  MPI_Wait (&request, &status)
  ```

- It blocks until a specified non-blocking send or receive operation has completed

- **Check** `MPI_Waitany`, `MPI_Waitall`

- Also check `MPI_Test`
Content Overview

• Basic concepts related to MPI

• Environment Management MPI routines

• Compiling and running MPI programs

• Types of communication
  – Point-to-Point communication routines
  – Collective communication routines

• Examples

• Summary
Collective Communication

• Defined as communication between > 2 processes
  – One-to-many, many-to-one, many-to-many

• All processes within the communicator group call the same collective communication function with matching arguments

• Collective communication routines are blocking

• The size of data sent must exactly match the size of data received
Collective Communication (synchronization)

• MPI_Barrier creates a barrier synchronization in a group
  – Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call

`MPI_Barrier (comm)`
Collective Communication (data movement)

- **MPI_Bcast** broadcasts (sends) a message from the process designated as "root" to all other processes in the group

  \[
  \text{MPI\_Bcast}(\&\text{buffer}, \text{count}, \text{datatype}, \text{root}, \text{comm})
  \]

- **MPI_Scatter** distributes distinct messages from a single source task to each task in the group

  \[
  \text{MPI\_Scatter}(\&\text{sendbuf}, \text{sendcnt}, \text{sendtype},
  \&\text{recvbuf}, \text{recvcnt}, \text{recvtype}, \text{root}, \text{comm})
  \]
Collective Communication (data movement)

- **MPI_Gather** is reverse of **MPI_Scatter** and gathers distinct messages from each task in the group to a single destination task

  ```
  MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)
  ```

- **MPI_Allgather** gathers data from all tasks in a group and distributes to all tasks

  ```
  MPI_Allgather(&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)
  ```
Collective Communication (collective computation)

- MPI_Reduce applies a reduction operation on all tasks in a group and places the result in one task

\[
\text{MPI\_Reduce} \left( \&\text{sendbuf}, \&\text{recvbuf}, \text{count}, \text{datatype}, \text{mpi\_red\_operation}, \text{root}, \text{comm} \right)
\]

<table>
<thead>
<tr>
<th>MPI Reduction Operation</th>
<th>Description</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>
</tbody>
</table>
Collective Communication (collective computation)

- **MPI_Allreduce** applies a reduction operation on all tasks in a group and passes the result to all tasks

  ```c
  MPI_Allreduce (&sendbuf,&recvbuf, count,
                 datatype, mpi_red_operation, comm)
  ```

- Many more functions that the audience might want to explore on their own, example, **MPI_Reduce_scatter**, **MPI_All_to_all**, **MPI_Scatterv**, **MPI_Gatherv**, ...
Content Overview

• Basic concepts related to MPI

• Environment Management MPI routines

• Compiling and running MPI programs

• Types of communication
  – Point-to-Point communication routines
  – Collective communication routines

• Examples

• Summary
Example with MPI_Reduce: reduce.c

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

int main(int argc, char *argv[])
{
    int i, value, total=0;
    int size, rank;

    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // assign value of 2^rank
    for (i=0, value=1; i<rank; i++) value *= 2;

    // call for the reduction
    MPI_Reduce (&value, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // report the result
    printf("Rank %d reports a value of %d and a total of %d\n", rank, value, total);

    // continued...
```
...and also MPI_Allreduce: reduce.c

// add a barrier so that output is separated, make it pretty...
MPI_Barrier(MPI_COMM_WORLD);
if (!rank) printf("\nFor MPI_Allreduce():\n\n");
MPI_Barrier(MPI_COMM_WORLD);

// allreduce doesn't require which rank to receive
MPI_Allreduce (&value, &total, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

// report the result
printf("Rank %d reports a value of %d and a total of %d\n", rank, value, total);

MPI_Finalize();
return 0;
}
Output from `reduce.c`

Rank 2 reports a value of 4 and a total of 0
Rank 1 reports a value of 2 and a total of 0
Rank 4 reports a value of 16 and a total of 0
Rank 3 reports a value of 8 and a total of 0
Rank 5 reports a value of 32 and a total of 0
Rank 6 reports a value of 64 and a total of 0
Rank 7 reports a value of 128 and a total of 0
Rank 0 reports a value of 1 and a total of 255

For MPI_Allreduce():

Rank 6 reports a value of 64 and a total of 255
Rank 0 reports a value of 1 and a total of 255
Rank 2 reports a value of 4 and a total of 255
Rank 3 reports a value of 8 and a total of 255
Rank 5 reports a value of 32 and a total of 255
Rank 7 reports a value of 128 and a total of 255
Rank 4 reports a value of 16 and a total of 255
Rank 1 reports a value of 2 and a total of 255
MPI_Bcast Example: broadcast.c

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

int main(int argc, char *argv[]){
    int i, rank, size;
    int root = 0, count = 5;
    int buffer[5];

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // root prepares data to broadcast
    if(rank == root) for(i=0; i<count; i++) buffer[i]=i;

    // broadcast the data
    MPI_Bcast(buffer, count, MPI_INT, root, MPI_COMM_WORLD);

    // all ranks report data received
    printf("Rank is: %d, Value at buffer[%d] is: %d \n", rank, count-1, buffer[count-1]);

    MPI_Finalize();
    return 0;
}
```
Output from `broadcast.c`

TACC: starting parallel tasks...
Rank is: 3, Value at buffer[4] is: 4
Rank is: 2, Value at buffer[4] is: 4
Rank is: 7, Value at buffer[4] is: 4
Rank is: 0, Value at buffer[4] is: 4
Rank is: 5, Value at buffer[4] is: 4
Rank is: 4, Value at buffer[4] is: 4
Rank is: 1, Value at buffer[4] is: 4
Rank is: 6, Value at buffer[4] is: 4
TACC: Shutting down parallel environment.

Note: Do not expect the output to be printed in any particular order. You might see jumbled up output.
Content Overview

• Basic concepts related to MPI

• Environment Management MPI routines

• Compiling and running MPI programs

• Types of communication
  – Point-to-Point communication routines
  – Collective communication routines

• Examples

• Summary
## Summary of Key MPI Routines

<table>
<thead>
<tr>
<th>C Function</th>
<th>Fortran Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init (argc, argv)</td>
<td>MPI_INIT (ierr)</td>
</tr>
<tr>
<td>MPI_Comm_size (comm, &amp;size)</td>
<td>MPI_COMM_SIZE (comm, size, ierr)</td>
</tr>
<tr>
<td>MPI_Comm_rank (comm, &amp;rank)</td>
<td>MPI_COMM_RANK (comm, rank, ierr)</td>
</tr>
<tr>
<td>MPI_Finalize ()</td>
<td>MPI_FINALIZE (ierr)</td>
</tr>
<tr>
<td>MPI_Send(&amp;buf, count, datatype, ...)</td>
<td>MPI_SEND (buf, count, datatype, ...)</td>
</tr>
<tr>
<td>MPI_Recv(&amp;buf, count, datatype, ...)</td>
<td>MPI_RECV (&amp;buf, count, datatype, ...)</td>
</tr>
<tr>
<td>MPI_Wtime()</td>
<td>MPI_WTIME()</td>
</tr>
</tbody>
</table>
Words of Caution!

• Not all applications can be parallelized
  – Analyze and understand the data dependencies in your application

• Not all parallelization result in speed-up (parallel slowdown)
  – Too much communication could be an overkill!

Note:
Total Execution Time = Computation Time + Communication Time + I/O time
References

https://computing.llnl.gov/tutorials/mpi/

http://www.mpi-forum.org

http://www.cs.usfca.edu/~peter/ppmpi/

http://www.mcs.anl.gov/research/projects/mpi/usingmpi/

http://geco.mines.edu/workshop/class2/examples/mpi/index.html
Next: OpenMP and shared memory parallelism

• What is OpenMP?

• How does OpenMP work?
  – Architecture
  – Fork-Join model of parallelism
  – Communication

• OpenMP Syntax
  – Compiler Directives
  – Runtime Library Routines
  – Environment variables

• What’s new? OpenMP 3.1
What is OpenMP?

- OpenMP stands for **Open Multi-Processing**
- An Application Programming Interface (API) for developing parallel programs for shared memory architectures
- Three primary components of the API are:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Standard specifies C, C++, and FORTRAN Directives & API
- [http://www.openmp.org/](http://www.openmp.org/) has the specification, examples, tutorials and documentation
OpenMP Fork-Join Parallelism

- Programs begin as a single process: master thread
- Master thread executes in serial mode until the parallel region construct is encountered
- Master thread creates a team of parallel threads (fork) that simultaneously execute statements in the parallel region
- After executing the statements in the parallel region, team threads synchronize and terminate (join) but master continues
How Do Threads Communicate?

• Every thread has access to “global” memory (shared)

• All threads share the same address space

• Threads communicate by reading/writing to the global memory

• Simultaneous updates to shared memory can create a race condition. Results change with different thread scheduling

• Use mutual exclusion to avoid data sharing - but don’t use too many because this will serialize performance
OpenMP Syntax

• Most of the constructs in OpenMP are compiler directives
  
  ```c
  #pragma omp construct [clause [,[,]clause]...] C
  !$omp    construct [clause [,[,]clause]...] F90
  ```

• Example
  
  ```c
  #pragma omp parallel num_threads(4) C
  !$omp    parallel num_threads(4) F90
  ```

• Function prototypes and types are in the file:
  
  ```c
  #include <omp.h> C
  use omp_lib F90
  ```

• 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.
Hello World OMP example: `omp_hello.c`

```c
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    int n_threads, tid;

    // Fork threads and have each report its number
    #pragma omp parallel private(tid, n_threads)
    {
        tid = omp_get_thread_num();
        n_threads = omp_get_num_threads();
        printf("Hello World from thread %d / %d \n", tid, n_threads);
    }

    return 0;
}
```
Compiling and running `omp_hello.c`

```bash
$ icc omp_hello.c -openmp -o omp_hello
$ export OMP_NUM_NUM_THREADS=5
$ ./omp_hello
Hello World from thread 0 / 5
Hello World from thread 1 / 5
Hello World from thread 2 / 5
Hello World from thread 4 / 5
Hello World from thread 3 / 5
```
Hello World OMP example: `omp_hello.c`

```c
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    int n_threads, tid;

    // Fork threads and have each report its number
    #pragma omp parallel num_threads(4) private(tid, n_threads)
    {
        tid = omp_get_thread_num();
        n_threads = omp_get_num_threads();
        printf("Hello World from thread %d / %d \n", tid, n_threads);
    }

    return 0;
}
```

Can also specify number of threads here.
OpenMP Constructs

OpenMP language “extensions”

- parallel control structures
- parallel control work sharing
- data environment
- synchronization
- runtime functions, env. variables

• governs flow of control in the program
  parallel directive

• distributes work among threads
  do/parallel do and Section directives

• specifies variables as shared or private
  shared and private clauses

• coordinates thread execution
  critical and atomic directives
  barrier directive

• Runtime functions
  omp_set_num_threads()
  omp_get_thread_num()
  OMP_NUM_THREADS
  OMP_SCHEDULE

• Env. Variable
  scheduling type
OpenMP Directives

- OpenMP directives are comments in source code that specify parallelism for shared memory machines
  
  FORTRAN : directives begin with the \texttt{!$OMP}, \texttt{C$OMP} or \texttt{*$OMP} sentinel.
  
  F90 : \texttt{!$OMP} free-format
  
  C/C++ : directives begin with the \texttt{# pragma omp} sentinel

- Parallel regions are marked by enclosing parallel directives
- Work-sharing loops are marked by parallel do/for

\textbf{Fortran}

\begin{verbatim}
 !$OMP parallel 
  ...
 !$OMP end parallel

 !$OMP parallel do
  DO ...
 !$OMP end parallel do
\end{verbatim}

\textbf{C/C++}

\begin{verbatim}
 # pragma omp parallel
  {...}

 # pragma omp parallel for
  for(){...}
\end{verbatim}
Parallel Region & Work-Sharing

Use OpenMP directives to specify Parallel Region & Work-Sharing constructs

```
Parallel
  Code block
    DO
    SECTIONS
    SINGLE
    CRITICAL
   End Parallel

Each Thread Executes
  Work Sharing
  Work Sharing
  One Thread (Work sharing)
  One Thread at a time

Parallel Region
  Work-Sharing
```

Parallel DO/for
Parallel SECTIONS
Parallel Regions

```c
#pragma omp parallel
{
    code block
    work(...);
}
```

Line 1  Team of threads formed at parallel region
Lines 3-4 Each thread executes code block and subroutine calls. No branching (in or out) in a parallel region
Line 5  All threads synchronize at end of parallel region (implied barrier)

Use the thread number to divide work among threads
Parallel Regions

1  !$OMP PARALLEL
2    code block
3    call work(...) 
4  !$OMP END PARALLEL

Line  1  Team of threads formed at parallel region.
Lines 2-3  Each thread executes code block and subroutine calls. No branching (in or out) in a parallel region.
Line  4  All threads synchronize at end of parallel region (implied barrier).

Use the thread number to divide work among.
Parallel Region & Number of Threads

• For example, to create a 10-thread Parallel region:

```c
double A[1000];
omp_set_num_threads(10);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID, A);
}
```

• Each thread redundantly executes the code within the structured block
• Each thread calls `foo(ID, A)` for `ID = 0` to `9`
OpenMP Parallel Constructs

**Replicated** : Work blocks are executed by all threads.

**Work-Sharing** : Work is divided among threads.

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

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

**Replicated**

```
I=N+1,2N
{code2}
I=2N+1,3N
{code2}
I=3N+1,4N
{code2}
I=1,N
{code2}
```

**Work-Sharing**

```
I=N+1,2N
{code1}
I=2N+1,3N
{code1}
I=3N+1,4N
{code1}
I=1,N
{code1}
```

**Combined**

```
I=N+1,2N
{code3}
I=2N+1,3N
{code3}
I=3N+1,4N
{code3}
I=1,N
{code3}
```
OpenMP Clauses

Clauses control the behavior of an OpenMP directive:

1. Data scoping (Private, Shared, Default)
2. Schedule (Guided, Static, Dynamic, etc.)
3. Initialization (e.g. COPYIN, FIRSTPRIVATE)
4. Whether to parallelize a region or not (if-clause)
5. Number of threads used (NUM_THREADS)
OpenMP Data Environment

• Data scoping clauses control the sharing behavior of variables within a parallel construct.
• These include *shared, private, firstprivate, lastprivate, reduction* clauses

Default variable scope:

1. Variables are shared by default
2. Global variables are shared by default
3. Automatic variables within subroutines called from within a parallel region are private (reside on a stack private to each thread), unless scoped otherwise
4. Default scoping rule can be changed with *default* clause
Private & Shared Data

**shared** - Variable is shared (seen) by all processors

**private** - Each thread has a private instance (copy) of the variable

Defaults: The for-loop index is private, all other variables are shared

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

All threads have access to the same storage areas for a, b, c, and n, but each loop has its own private copy of the loop index, i
Private Data Example

• In the following loop, each thread needs its own private copy of temp
• If temp were shared, the result would be unpredictable since each thread would be writing and reading to/from the same memory location

```cpp
#pragma omp parallel for shared(a,b,c,n) private(temp,i)
for (i=0; i<n; i++){
    temp = a[i] / b[i];
    c[i] = temp + cos(temp);
}
```

• 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)` would copy the global temp value to each stack’s temp.
Reduction

• Operation that combines multiple elements to form a single result
• A variable that accumulates the result is called a reduction variable
• In parallel loops reduction operators and variables must be declared
  
  ```c
  float asum, aprod;
  asum = 0.;
  aprod = 1.;
  #pragma omp parallel for reduction(+:asum) reduction(*:aprod)
  for (i=0; i<n; i++){
      asum = asum + a[i];
      aprod = aprod * a[i];
  }
  ```
  
  Each thread has a private `asum` and `aprod`, initialized to the operator’s identity
• After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction
Synchronization

• Synchronization is used to impose order constraints and to protect access to shared data

• High-Level Synchronization
  – critical
  – atomic
  – barrier
  – ordered

• Low-Level Synchronization
  – locks
Synchronization: Critical/Atomic Directives

• When each thread must execute a section of code serially the region must be marked with `critical/end critical` directives

• Use the `#pragma omp atomic` directive if executing only one operation serially

```c
#pragma omp parallel shared(sum,x,y)
...
#pragma omp critical
{
  update(x);
  update(y);
  sum=sum+1;
}
...
!$OMP END PARALLEL
```

```
#pragma omp parallel shared(sum)
...
{
  #pragma omp atomic
  sum=sum+1;
  ...
}
```

Master Thread

CRITICAL section or atomic operations
Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive

```c
#pragma omp parallel shared (A, B, C) private(id)
{
    id = omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
    for (i=0; i<N; i++){
        C[i] = big_calc3(i, A);
    }
    #pragma omp for nowait
    for (i=0; i<N; i++){
        B[i] = big_calc2(C, i);
    }
    A[id] = big_calc4(id);
}
```

Implicit barrier

No implicit barrier due to nowait

Implicit barrier
Mutual Exclusion: Lock Routines

When each thread must execute a section of code serially locks provide a more flexible way of ensuring serial access than **CRITICAL** and **ATOMIC** directives.

call OMP_INIT_LOCK(maxlock)
!$OMP PARALLEL SHARED(X,Y)
...
call OMP_set_lock(maxlock)
call update(x)
call OMP_unset_lock(maxlock)
...
!$OMP END PARALLEL
call OMP_DESTROY_LOCK(maxlock)
Synchronization: Ordered

- The ordered region executes in the sequential order

```c
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:countVal)
for (i=0; i<N; i++){
    tmp = foo(i);
    #pragma omp ordered
    countVal+= consume(tmp);
}
```
• When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.

• By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```c
#pragma omp parallel
{
    #pragma omp for nowait
    {
        for (i=0; i<n; i++)
            {work(i);}
    }
    #pragma omp for schedule(dynamic,k)
    {
        for (i=0; i<m; i++)
            {x[i]=y[i]+z[i];}
    }
}
```
## Runtime Library Routines

<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 team, N</td>
</tr>
<tr>
<td><code>omp_get_thread_num()</code></td>
<td>Thread ID {0 -&gt; N-1}</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 thread executing</td>
</tr>
<tr>
<td><code>omp_set_num_threads(#)</code></td>
<td>Set the number of threads in the team</td>
</tr>
<tr>
<td><code>omp_get_dynamic()</code></td>
<td>True if dynamic threading is on</td>
</tr>
<tr>
<td><code>omp_set_dynamic()</code></td>
<td>Set state of dynamic threading (true/false)</td>
</tr>
</tbody>
</table>
## Environment Variables

<table>
<thead>
<tr>
<th>variable</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_NUM_THREADS int_literal</td>
<td>Set to default no. of threads to use</td>
</tr>
<tr>
<td>OMP_SCHEDULE “schedule[, chunk_size]”</td>
<td>Control how “omp for schedule(RUNTIME)” loop iterations are scheduled</td>
</tr>
<tr>
<td>OMP_DYNAMIC</td>
<td>TRUE/FALSE for enable/disable dynamic threading</td>
</tr>
</tbody>
</table>
NUM_THREADS clause

• Use the NUM_THREADS clause to specify the number of threads to execute a parallel region

```
#pragma omp parallel num_threads(scalar int expression)
{
    <code block>
}
```

where scalar integer expression must evaluate to a positive integer

• NUM_THREADS supersedes the number of threads specified by the OMP_NUM_THREADS environment variable or that set by the OMP_SET_NUM_THREADS function
While OpenMP 3.1 supports nested parallelism, many implementations may ignore the nesting by serializing the inner parallel regions
References

• http://www.openmp.org/

• *Parallel Programming in OpenMP*, by Chandra, Dagum, Kohr, Maydan, McDonald, Menon

• *Using OpenMP*, by Chapman, Jost, Van der Pas (OpenMP2.5)

• https://computing.llnl.gov/tutorials/openMP/