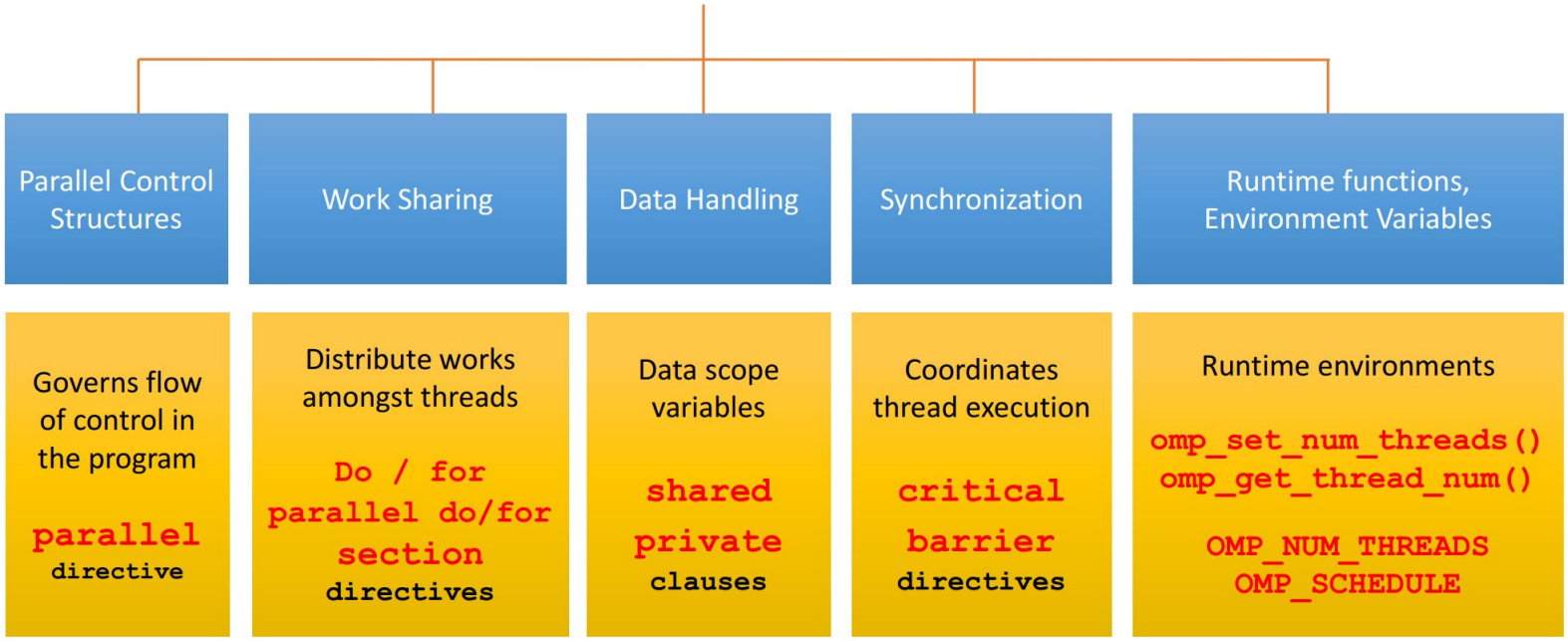
Parallel & Distributed Computing
Lecture 14:
Distributed Memory Programming

Farhad M. Riaz

Farhad.Muhammad@numl.edu.pk

Department of Computer Science
NUML, Islamabad

OpenMP Language Extensions



OpenMP Constructs

- Parallel region

  #pragma omp parallel

- Worksharing

  #pragma omp for

  #pragma omp sections

- Data Environment

  #pragma omp parallel shared/private (...)

- Synchronization

  #pragma omp barrier

  #pragma omp critical

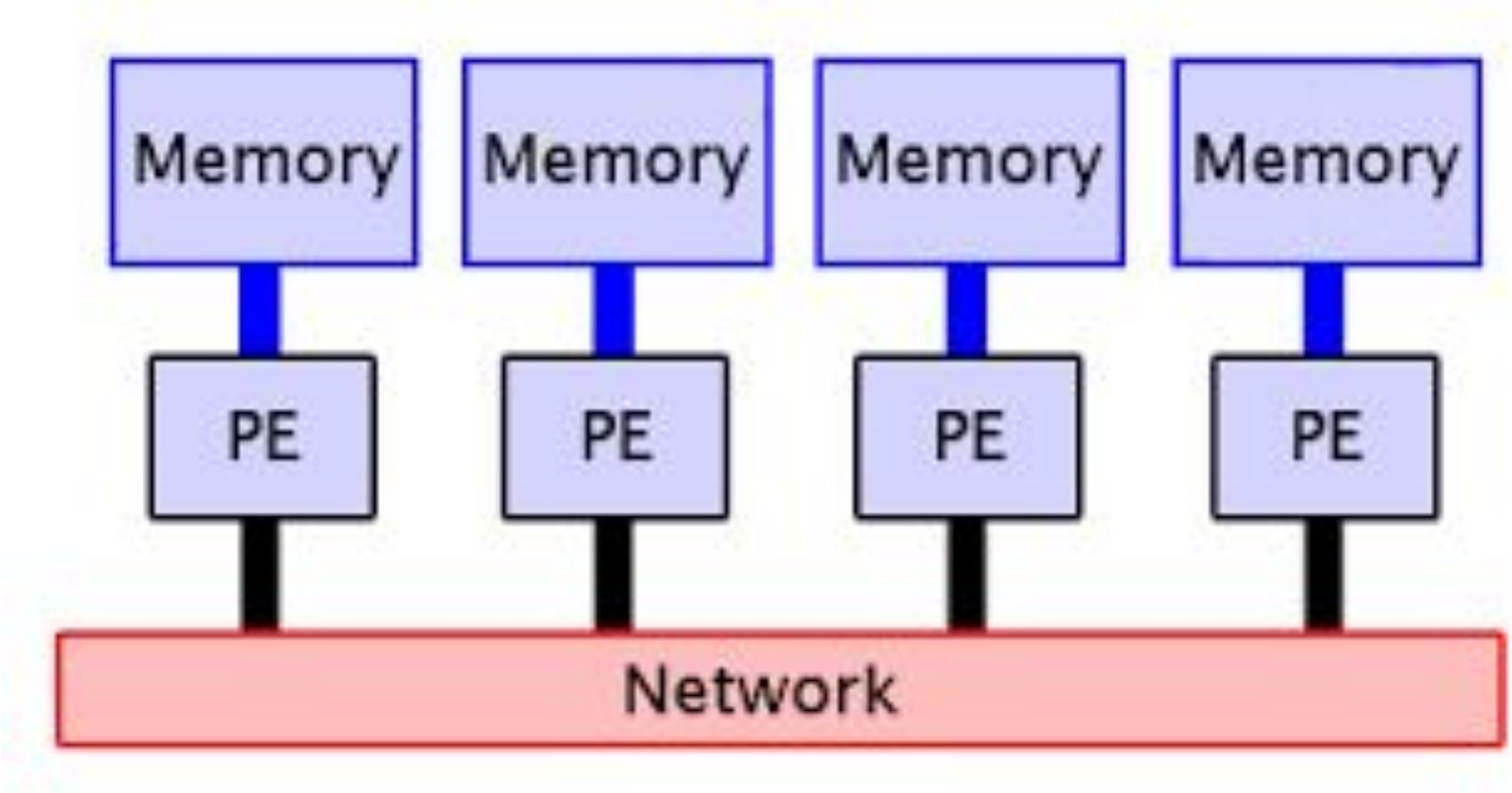
Work Sharing Example

Example ploop.1.c (pre_omp_3.0)

```
void simple(int n, float *a, float *b)
{
    int i;

#pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Distributed Memory Systems



Message Passing

- A process is a program counter and address space.
- Message passing is used for communication among processes.
- Inter-process communication:
  - Type:
    Synchronous / Asynchronous
  - Movement of data from one process's address space to another's

Synchronous vs asynchronous

- A synchronous communication is not complete until the message has been received.

- An asynchronous communication completes as soon as the message is on the way.

Blocking vs. Non-blocking

- Blocking, means the program will not continue until the communication is completed.

- Non-Blocking, means the program will continue, without waiting for the communication to be completed.

Background on MPI

- MPI - Message Passing Interface
  - Library standard defined by committee of vendors, implementers, and parallel programmer
  - Used to create parallel SPMD programs based on message passing
- Available on almost all parallel machines in C and Fortran
- About 125 routines including advanced routines
- 6 basic routines

MPI Implementations

- Most parallel machine vendors have optimized versions
- Others:
  - http://www-unix.mcs.anl.gov/mpi/mpich/
  - GLOBUS:
  - http://www3.niu.edu/mpi/
  - http://www.globus.org

Key Concepts of MPI

- Used to create parallel SPMD programs based on message passing
- Normally the same program is running on several different nodes
- Nodes communicate using message passing

Advantages of Message Passing

- **Universality**
- **Expressivity**
- **Ease of debugging**

Advantages of Message Passing

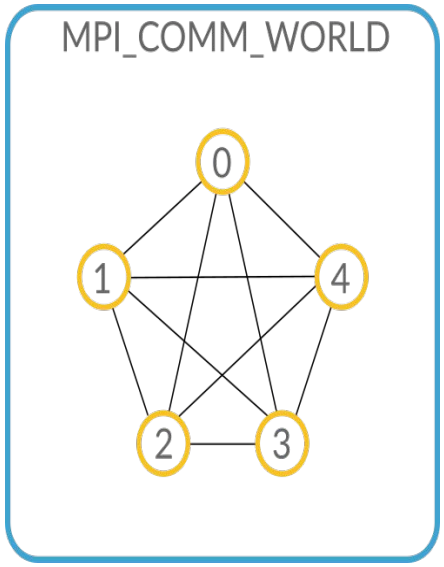
- Performance:
  - This is the most compelling reason why MP will remain a permanent part of parallel computing environment
  - As modern CPUs become faster, management of their caches and the memory hierarchy is the key to getting most out of them
  - MP allows a way for the programmer to explicitly associate specific data with processes and allows the compiler and cache management hardware to function fully
  - Memory bound applications can exhibit super-linear speedup when run on multiple PEs compare to single PE of MP machines

Include files

- The MPI include file
  - mpi.h
- Defines many constants used within MPI programs
- In C defines the interfaces for the functions
- Compilers know where to find the include files

Communicators

- A parameter for most MPI calls
- A collection of processors working on some part of a parallel job
- MPI_COMM_WORLD is defined in the MPI include file as all of the processors in your job
- Can create subsets of MPI_COMM_WORLD
- Processors within a communicator are assigned numbers 0 to n-1



Data types

- When sending a message, it is given a data type
- Predefined types correspond to "normal" types
  - MPI_REAL , MPI_FLOAT - Fortran real and C float respectively
  - MPI_DOUBLE_PRECISION , MPI_DOUBLE - Fortan double precision and C double repectively
  - MPI_INTEGER and MPI_INT - Fortran and C integer respectively
- User can also create user-defined types

Minimal MPI program

```
#include <mpi.h> /* the mpi include file */

/* Initialize MPI */
ierr=MPI_Init(&argc, &argv);

/* How many total PEs are there */
ierr=MPI_Comm_size(MPI_COMM_WORLD, &nPEs);

/* What node am I (what is my rank? */
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &iam);
...
ierr=MPI_Finalize();
```

MPI “Hello, World”

- A parallel hello world program
  - Initialize MPI
  - Have each node print out its node number
  - Quit MPI

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

int main(int argc, char** argv){
    int process_Rank, size_Of_Cluster;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

    printf("Hello World from process %d of %d\n", process_Rank, size_Of_Cluster);

    MPI_Finalize();
    return 0;
}
```

```
Hello World from process 3 of 4
Hello World from process 2 of 4
Hello World from process 1 of 4
Hello World from process 0 of 4
```

C/MPI version of “Hello, World”

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>
int main(int argc, char *argv[ ])
{
   int myid, numprocs;

   MPI_Init (&argc, &argv) ;
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs) ;
   MPI_Comm_rank(MPI_COMM_WORLD, &myid) ;
   printf(“Hello from %d\n”, myid) ;
   printf(“Numprocs is %d\n”, numprocs) ;

   MPI_Finalize():
}
```

Basic Communications in MPI

- Data values are transferred from one processor to another
  - One process sends the data
  - Another receives the data
- Synchronous
  - Call does not return until the message is sent or received
- Asynchronous
  - Call indicates a start of send or receive operation, and another call is made to determine if call has finished

Synchronous Send

- MPI_Send: Sends data to another processor
- Use MPI_Receive to "get" the data

```
MPI_Send(&buffer,count,datatype,
destination,tag,communicator);
```

- Call blocks until message on the way

```
void* message;           //Address for the message you are sending.
int count;               //Number of elements being sent through the addre
MPI_Datatype datatype;   //The MPI specific data type being passed through
int dest;                //Rank of destination process.
int tag;                 //Message tag.
MPI_Comm comm;           //The MPI Communicator handle.
```

```
void* message;           //Address to the message you are receiving.
int count;               //Number of elements being sent through the addr
MPI_Datatype datatype;   //The MPI specific data type being passed throug
int from;                //Process rank of sending process.
int tag;                 //Message tag.
MPI_Comm comm;           //The MPI Communicator handle.
MPI_Status* status;      //Status object.
```

MPI_Send

- Buffer: The data
- Count : Length of source array (in elements, 1 for scalars)
- Datatype : Type of data, for example MPI_DOUBLE_PRECISION (fortran) , MPI_INT (C), etc
- Destination : Processor number of destination processor in communicator
- Tag : Message type (arbitrary integer)
- Communicator : Your set of processors
- Ierr : Error return (Fortran only)

Synchronous Receive

- Call blocks until message is in buffer
  - **MPI_Recv(&buffer,count, datatype, source, tag, communicator, &status);**

- Status - contains information about incoming message
  - **MPI_Status status;**

Status

- status is a structure of type MPI_Status which contains three fields MPI_SOURCE, MPI_TAG, and MPI_ERROR
- status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code respectively of the received message

MPI_Recv

- Buffer: The data
- Count : Length of source array (in elements, 1 for scalars)
- Datatype : Type of data, for example : MPI_DOUBLE_PRECISION, MPI_INT, etc
- Source : Processor number of source processor in communicator
- Tag : Message type (arbitrary integer)
- Communicator : Your set of processors
- Status: Information about message
- Ierr : Error return (Fortran only)

Basic MPI Send and Receive

- A parallel program to send & receive data
  - Initialize MPI
  - Have processor 0 send an integer to processor 1
  - Have processor 1 receive an integer from processor 0
  - Both processors print the data
  - Quit MPI

Simple Send & Receive Program

```
#include <stdio.h>
#include "mpi.h"
#include <math.h>

   /*********************************************************************
   **
This is a simple send/receive program in MPI
*********************************************************************/
int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Simple Send & Receive Program (cont.)

```
  tag=1234;
  source=0;
  destination=1;
  count=1;
  if(myid == source) {
     buffer=5678;
     MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
     printf("processor %d  sent %d\n",myid,buffer);
  }
  if(myid == destination) {
     MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
      printf("processor %d  got %d\n",myid,buffer);
  }
  MPI_Finalize();
}
```

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv){
    int process_Rank, size_Of_Cluster, message_Item;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size_Of_Cluster);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_Rank);

    if(process_Rank == 0){
        message_Item = 42;
        printf("Sending message containing: %d\n", message_Item)
    }
    else if(process_Rank == 1){
        printf("Received message containing: %d\n", message_Item)
    }
    MPI_Finalize();
    return 0;
}
```

The 6 Basic C MPI Calls

- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- The 6 basic calls inC MPI are:
  - MPI_Init(&argc, &argv)
  - MPI_Comm_rank(MPI_COMM_WORLD, &myid)
  - MPI_Comm_Size(MPI_COMM_WORLD, &numprocs)
  - MPI_Send(&buffer, count,MPI_INT,destination, tag, MPI_COMM_WORLD)
  - MPI_Recv(&buffer, count, MPI_INT,source,tag, MPI_COMM_WORLD, &status)
  - MPI_Finalize()

Further Reading

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

That's all for today!!