CMSC 838T – Lecture 8

- **Parallel programming**
  - Examine parallel programming paradigms & languages
  - Case study of OpenMP and MPI

- Fork-join paradigm (OpenMP)
- SPMD paradigm (MPI)

---

**Parallel Programming Languages**

- **Why parallel programming languages?**
  - Automatic parallelization of sequential programs is difficult
  - Easier to write programs to explicitly exploit parallelism

- **Goals**
  - Easy to program
    - Decomposition, mapping, communication, synchronization
    - Avoids errors – data races, deadlocks, buffer overflows
  - Flexible paradigm & architecture independent
    - Supports wide range of parallel applications
    - Runs on a variety of parallel architectures
  - Performance
    - Able to achieve good performance on parallel architecture
    - Simple programming model to sustain performance
Parallel Programming Languages

◆ **Goals conflict!**
◆ **In practice, languages fall into two camps**
  - Shared memory paradigm
    - Specify parallelism
    - Relatively easy to program
    - Runs efficiently on shared-memory architectures
    - Some can run on distributed memory architectures (efficiency varies, depending on application)
  - Distributed memory paradigm
    - Specify parallelism & interprocessor communication
    - More difficult to program
    - Runs on both shared & distributed-memory architectures

Shared memory with explicit threads
  - Pthreads, Java threads

Shared memory with implicit threads
  - Data parallel (SIMD) – Fortran 90, HPF
  - Parallel loops / tasks (MIMD) – OpenMP, UPC

Distributed memory with explicit communication
  - MPI, SHMEM, Distributed Java

Distributed memory with special global accesses
  - Co-Array Fortran, Global Arrays, UPC
Parallel Programming Overview

- **Motivation**
- **Shared memory paradigms**
  - Explicit threads
  - Implicit threads
- **Distributed memory paradigms**
  - Explicit messages
  - Implicit messages (special nonlocal accesses)
- **Comparisons**
- **Case study**
  - OpenMP
  - MPI

Pthreads, Java Threads

- **Characteristics**
  - All memory shared (except local thread variables)
  - Parallelism: explicit thread creation
  - Underlying implementation for many paradigms
  - Medium-grain parallelism, not available on clusters
- **Example**
  ```c
  for (i=0; i<PROC; i++)  { pthread_create(&p[i], NULL, runThr, Init[i]); }
  for (i=0; i<PROC; i++)  { pthread_join(&p[i], NULL); }
  ```
  ```c
  void * runThr(void * args) {
    myThread = *((int *)args); /* init my thread */
    allThreadBarrier();        /* wait for all threads to initialize */
    compute();                 /* do work in parallel */
  }
  ```
Parallel Programming Overview

- **Motivation**
- **Shared memory paradigms**
  - Explicit threads
  - Implicit threads
- **Distributed memory paradigms**
  - Explicit messages
  - Implicit messages (special nonlocal accesses)
- **Comparisons**
- **Case study**
  - OpenMP
  - MPI

---

OpenMP

- **Characteristics**
  - Both local & shared memory (depending on directives)
  - Parallelism : directives for parallel loops, functions
  - Compilers convert programs into Pthreads
  - Not available on clusters
- **Example**

```c
#pragma omp parallel for private(i)
for (i=0; i<NUPDATE; i++) {
    int ran = random();
    table[ ran & (TABSIZE-1) ] ^= stable[ ran >> (64-LSTSIZE) ];
}
```
FORTRAN 90

◆ Characteristics
- Extension of 1D vector operations
- Arrays treated as single object
- Operations on array performed in parallel (SIMD, data-parallel)
- Many array-based intrinsic operators and functions

◆ Examples
real, dimension(100, 100) :: x, y, z
z = x + y ! z(i, j) = x(i, j) + y(i, j)
x = 1.0 ! x(i, j) = 1.0
b = x .eq. y ! if x(i, j) == y(i, j) then b(i, j) = .true.
  else b(i, j) = .false.
where (x>1) y=1 ! if x(i, j) >1 then y(i, j) = 1
i = sum (x)! i = scalar sum of all elements in x

CMSC 838T – Lecture 8

High Performance Fortran (HPF)

◆ Characteristics
- Both local & shared memory
- Fortran 90 + data distribution directives
- Parallelism : data-parallelism, directives for loops, functions
- Compilers generate explicit communication

◆ Example
REAL X(N,N), Y(N,N)
!HPF$ DISTRIBUTE X(BLOCK, *)
!HPF$ ALIGN (: , :) WITH X(: , :) :: Y
FORALL (I = 1,N-1, J = 1,N-1) &
  Y(I,J) = 0.25 * (X(I-1,J-1) + X(I-1,J+1) + X(I+1,J-1) + X(I+1,J+1))
X = Y

CMSC 838T – Lecture 8
Parallel Programming Overview

- **Motivation**
- **Shared memory paradigms**
  - Explicit threads
  - Implicit threads
- **Distributed memory paradigms**
  - Explicit messages
  - Implicit messages (special nonlocal accesses)
- **Comparisons**
- **Case study**
  - OpenMP
  - MPI

**MPI, SHMEM, Distributed Java**

- **Characteristics**
  - All memory local
  - Parallelism: same program on multiple nodes (SPMD)
  - Explicit communication for non-local memory access
    - MPI: matching send / receive
    - SHMEM: one-sided put / get
    - Distributed Java: asynchronous pipes
  - Large collective communication library (MPI, SHMEM)
- **Example**
  
  if (send) MPI_Send (BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
  
  if (recv) MPI_Recv (BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR); 

  MPI_Alltoall (LOCPT, SIZE, MPI_INT, GLOBPT, SIZE, MPI_INT, MPI_COMM_WORLD);
Parallel Programming Overview

◆ Motivation
◆ Shared memory paradigms
  - Explicit threads
  - Implicit threads
◆ Distributed memory paradigms
  - Explicit messages
  - Implicit messages (special nonlocal accesses)
◆ Comparisons
◆ Case study
  - OpenMP
  - MPI

Co-Array Fortran

◆ Characteristics
  - Local memory, shared global arrays
  - Parallelism: single program on multiple nodes (SPMD)
  - Provides illusion of shared multidimensional arrays
    - Extra array dimension specifying processor location
  - Simple approach to specifying non-local accesses
  - User responsible for specifying processor
◆ Example

```fortran
REAL, DIMENSION (N) [*] :: X, Y
X(:) = Y(:) [ MOD(THIS_IMAGE + 1, NUM_IMAGES) ]
```

CMSC 838T – Lecture 8
Global Arrays

◆ Characteristics
  - Local memory, shared global arrays
  - Parallelism: single program on multiple nodes (SPMD)
  - Provides illusion of shared multidimensional arrays
  - Library routines
    - Copy rectangular shaped data in & out of global arrays
    - Scatter / gather / accumulate operations on global array
  - Designed to be more restrictive, easier to use than MPI

◆ Example
  
  NGA_Access(g_a, lo, hi, &table, &ld);
  for (j = 0; j < PROCS; j++) {
    for (i = 0; i < counts[j]; i++) {
      table[index-lo[0]] ^= stable[copy[i] >> (64-LSTSIZE)];
    }
  }
  NGA_Release_update(g_a, lo, hi);

UPC

◆ Characteristics
  - Local memory, shared arrays accessed by global pointers
  - Parallelism: single program on multiple nodes (SPMD)
  - Provides illusion of shared one-dimensional arrays
  - Features
    - Data distribution declarations for arrays
    - Cast global pointers to local pointers for efficiency
    - One-sided communication routines (memput / memget)
  - Compilers translate global pointers, generate communication

◆ Example
  
  shared int *x, *y, z[100];
  upc_forall (i = 0; i < 100; j++) {
    z[i] = *x++ *y++;
  }
Parallel Programming Overview

◆ Motivation

◆ Shared memory paradigms
  - Explicit threads
  - Implicit threads

◆ Distributed memory paradigms
  - Explicit messages
  - Implicit messages (special nonlocal accesses)

◆ Comparisons

◆ Case study
  - OpenMP
  - MPI

Comparison – Programmability

◆ Shared memory
  - Easiest
  - User inserts parallelism, data distribution directives

◆ Shared memory with explicit threads
  - Easy, but lower level
  - Users encapsulate parallelism in functions

◆ Distributed memory with special global accesses
  - Medium
  - Users insert directives, differentiate local / global accesses

◆ Distributed memory with explicit communication
  - Difficult, and low-level
  - Users insert explicit communication for non-local accesses
Comparison – Portability & Performance

- **Shared memory**
  - Portability – poor to medium
  - Performance – poor to excellent (depending on architecture)
- **Shared memory with explicit threads**
  - Portability – poor
  - Performance – excellent
- **Distributed memory with special global accesses**
  - Portability – moderate
  - Performance – poor to medium (depending on application)
- **Distributed memory with explicit communication**
  - Portability – excellent
  - Performance – excellent

Comparison – Sources of Error

- **Shared memory with (implicit or explicit) threads**
  - Data races (very difficult to find)
    - Variable read / written at wrong time
    - Output depends on thread execution order
    - Example
      - \( X = 1; \quad Y = X; \)
      - (value of \( Y \) varies)
      - \( Y = X; \quad X = 2; \)
  - Deadlock
    - Threads frozen waiting for lock synchronization
- **Distributed memory with explicit communication**
  - Buffer overflow
    - Too many messages received by thread
    - Unable to buffer / process messages exceeding limit
    - Thread freezes waiting for dropped messages
Parallel Programming Overview

- **Motivation**
- **Shared memory paradigms**
  - Explicit threads
  - Implicit threads
- **Distributed memory paradigms**
  - Explicit messages
  - Implicit messages (special nonlocal accesses)
- **Comparisons**
- **Case study**
  - OpenMP
  - MPI

Calculating $\pi$ – Sequential Program

- **Compute an approximation to $\pi$**
  - Using numerical integration
  - Find the area under the curve $4/(1+x^2)$ between 0 and 1

\[ \int_0^1 \frac{4}{1 + x^2} \, dx = \pi \]
Calculating $\pi$ – Sequential Program

```c
int num_steps = 1000;
double width;
void main ()
{
    int i;
    double x, pi, sum = 0.0;
    width = 1.0 / (double) num_steps;
    for (i=1; i <= num_steps; i++) {
        x = (i-0.5)* width;
        sum = sum + 4.0 / (1.0+x*x);
    }
    pi = sum * width;
}
```

Shared-Memory Paradigm – OpenMP

- **Characteristics**
  - Not a full parallel language, but a language extension
  - A set of standard compiler directives and library routines
  - Used to create parallel Fortran, C and C++ programs
  - Usually used to parallelize loops
  - Standardizes last 15 years of SMP practice

- **Implementation**
  - Compiler directives using `#pragma omp <directive>`
  - Parallelism can be specified for regions & loops
  - Data can be
    - Private – each processor has local copy
    - Shared – single copy for all processors
OpenMP – Programming Model

- **Fork-join parallelism (restricted form of MIMD)**
  - Normally single thread of control (master)
  - Worker threads spawned when parallel region encountered
  - Barrier synchronization required at end of parallel region

OpenMP – Example Parallel Region

- **Task level parallelism – #pragma omp parallel { ... }**

```c
double a[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int id = omp_thread_num();
    foo(id,a);
}
printf(“all done 
”);
```
OpenMP – Example Parallel Loop

- **Loop level parallelism** – #pragma omp parallel for
  - Loop iterations are assigned to threads, invoked as functions

```c
#pragma omp parallel
{
  int id, i, nthreads, start, end;
  id = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  start = id * N / nthreads;  // assigning
  end = (id+1) * N / nthreads;  // work
  for (i=start; i<end; i++) {
    foo(i);
  }
}
```

Parallel Programming Overview

- **Motivation**
- **Shared memory paradigms**
  - Explicit threads
  - Implicit threads
- **Distributed memory paradigms**
  - Explicit messages
  - Implicit messages (special nonlocal accesses)
- **Comparisons**
- **Case study**
  - OpenMP
  - MPI
Distributed-Memory Paradigm – MPI

◆ Message Passing Interface (MPI)
  - MPI is a message-passing library specification
  - Extended message-passing model
  - Not a specific implementation or product
◆ Most general & popular parallel paradigm in use
  - Started as informal standard in 1994
  - Both public and vendor-specific implementations
  - Portable & efficient
◆ Paradigm
  - Message passing

MPI Programming Overview

◆ Creating parallelism
  - SPMD Model
◆ Communication between processors
  - Point-to-point
  - Collective
◆ Synchronization
  - Point-to-point synchronization is done by message passing
  - Global synchronization done by collective communication
MPI Language Binding – C and Fortran

- **MPI is language independent**
  - Has “bindings” for C, Fortran, other languages
- **In C:**
  - mpi.h must be included
  - MPI functions return error codes or MPI_SUCCESS
- **In Fortran:**
  - mpif.h must be included
  - All MPI calls are subroutines, with a place for the return code in the last argument.
- **C++ bindings, and Fortran-90 issues, are part of MPI-2**

SPMD Programming Model

- **SPMD (Single Program Multiple Data) model**
  - Each processor has a copy of the same program
  - All run them at their own rate
  - May take different paths through the code
- **Process-specific control through variables like**
  - My process number
  - Total number of processors
- **Processors may synchronize explicitly**
  - Call synchronization library functions
SPMD Programming Model

```c
my_id=GetID();
if (my_id == 1)
    Execute Task 1;
if (my_id == 2)
    Execute Task 2;
...
if (my_id == n)
    Execute Task n;
```

◆ **MPI processes differentiate themselves**
  - Using MPI library calls

◆ **Example**

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf("I am process %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```
Message-Passing Abstraction

- Sender specifies buffer to be transmitted and receiving process
- Receiver specifies sending process and application storage to receive into
- Optional tag on send and matching rule on receive
- In simplest form, the blocking send/recv match achieves a pairwise synchronization event and a memory-to-memory copy
- Overhead: copying, buffer management, communication delay

MPI Message Passing – Point-to-Point

- Passing messages with send() and recv() library calls
- Need to specify
  - How will data be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?
MPI Message Passing – Send

MPI_Send (start, count, datatype, dest, tag, comm)
- start a pointer to the start of the data
- count the number of elements to be sent
- datatype the type of the data
- dest the ID of the destination process
- tag the tag on the message for matching
- comm the communicator to be used

◆ Semantics (when function returns)
- Data has been delivered to “system”
  • i.e., copied to system buffer
- Data structure (start…start+count) can be reused
- Destination process X may not have received message!
  • X stores data to system buffer until matching recv() called

MPI Message Passing – Receive

MPI_Recv(start, count, datatype, src, tag, comm, status)
- start a pointer to the start of the data destination
- count the number of elements to be received
- datatype the type of the data
- src the ID of the source process
- tag the tag on the message for matching
- comm the communicator to be used
- status the place to put status information

◆ Semantics (when function called)
- Wait until send() call with matching src / tag is received
- May freeze if matching message is not received!

◆ Semantics (when function returns)
- Data from send() has been received and stored into start
  • i.e., copied from system buffer
Describing Data – MPI Datatypes

- Data in MPI message is described by a triple
  - \( \langle \text{address}, \text{count}, \text{datatype} \rangle \)

- An MPI datatype is recursively defined as:
  - Predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - A contiguous array of datatypes
  - A strided block of datatypes
  - An indexed array of blocks of datatypes
  - An arbitrary structure of datatypes

- Supports using MPI on heterogeneous architectures

MPI Example – Calculating \( \pi \)

```c
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
  int n, myid, numprocs, i;
  double mypi, pi, h, sum, x;
  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  if (myid == 0) {
    printf("Enter the number of intervals:\n");
    scanf("%d",&n);
    for(i=1;i<numprocs;i++)
      MPI_Send(&n,1,MPI_INT,i,0,MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(&n,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
  }
```

CMSC 838T – Lecture 8
MPI Example – Calculating $\pi$ (cont.)

```c
h = 1.0 / (double) n; sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;
if (myid != 0)
    MPI_Send(&mypi,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
else {
    pi = mypi;
    for(i=1;i<numprocs;i++)
        MPI_Recv(&mypi,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&status);
    pi += mypi;
}
printf("pi is approximately \%.16fn",pi);
```

MPI_Finalize();
return 0;
}
```

Message Passing – Collective Communication

- **Collective communication**
  - Routines that send message(s) to a group of processes or receive message(s) from a group of processes
  - Potentially more efficient than point-to-point communication
  - Not absolutely necessary

- **Examples**
  - Broadcast
  - Scatter
  - Gather
  - Reduction
  - All-to-all
  - Barrier
Collective Communication – Broadcast

- Sends data from root to all others in a group
  - Must be called by all processes in group with same arguments
  ```
  MPI_Bcast(data, count, datatype, source, comm);
  ```

Collective Communication – Scatter

- Sends each element of array in root to separate process
  - Contents of ith location of array sent to ith process
  ```
  MPI_Scatter(send_data, send_count, send_type, recv_data, recv_count, recv_type, root, comm)
  ```
Collective Communication – Gather

- **Collects data from set of processes**
- Store the data in an array

\[
\text{MPI\_Gather}(send\_data, send\_count, send\_type, \\recv\_data, recv\_count, recv\_type, root, comm)
\]

Collective Communication – Reduce

- **Combines data from all processes in group**
- Performs (associative) reduction operation (SUM, MAX)
- Returns the data to one process

\[
\text{MPI\_Reduce}(\text{operand}, \text{result}, \text{count}, \text{datatype}, \\text{operation}, \text{dest}, \text{comm})
\]
Collective Communication – Calculating π

```c
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    if (myid == 0) {
        printf("Enter the number of intervals");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
        printf("pi is approximately %.16f\n",pi);
    MPI_Finalize();
    return 0;
}
```
Parallel Programming Languages

◆ Summary
  - Variety of programming paradigms
    - Shared memory – OpenMP
    - Distributed memory – MPI
  - Different strengths, weaknesses

◆ Still searching for
  - Shared-memory paradigm that efficiently executes on distributed-memory parallel architectures
  - May not be possible given underlying architectures

Multiprogramming vs. Parallel Programming

◆ Multiprogramming
  - Multiple, unrelated, instruction streams
    - Execute on single or multiple processors
    - Overlap execution to hide latency, fully utilize resources
  - Increases throughput (reduce execution time for all programs)
  - Does not reduce execution time of single program

◆ Parallel programming
  - Multiple, related, interacting instruction streams
    - Execute on multiple processors
    - Incur overhead, underutilize resources
  - Reduce execution time of single program
Bioinformatics Applications

- **Current practice**
  - Usually many unrelated tasks
  - Tasks are multiprogrammed on collection of servers
- **NCBI example**
  - NCBI maintains cluster of 80+ PCs for GenBank
  - Web server receives request to “blast” sequences X, Y, Z…
  - Farms out individual requests to separate PCs
  - Collects answer and create web page with result
- **Can viewed as distributed parallel application**

- **As size of sequence databases grow**
  - May need to exploit parallelism for individual applications

Bioinformatics and Parallel Computing

- **Targets for high performance computing**
  - Sequence alignment / search embarrassing parallel
  - Protein structure prediction fine-grain parallel
  - Protein docking embarrassing parallel
  - Gene expression analysis parallel
  - Phylogenetic analysis embarrassing parallel
- **Distributed computing principal model**
  - Coarse-grain task-level parallelism
  - Data locality / distribution important
- **Open question**
  - What fields of bioinformatics will benefit…
  - …if parallel computing enables more precise algorithms