LECTURE 5: MESSAGE-ORIENTED COMMUNICATION

Lecture Contents

- Message Passing primitives: Send/Receive
- Synchronous & Asynchronous Message Passing
- Types of Processes in Message Passing
- Examples
  - Synchronous Network of Filters: Sieve of Eratosthenes
  - Asynchronous Heartbeat Algorithm for Network Topology
  - Synchronous Heartbeat Algorithm for Parallel Sorting
- Introduction to Parallel Programming with Message Passing:
  - In MPI
  - In OpenMP
  - Using both together
SECTION 5.1: BASICS OF MESSAGE PASSING

Introduction to Message Passing

- Up to now concurrency constructs (critical sections, semaphores, monitors) have been based on shared memory systems.
- However with network architectures & distributed systems in which processors are only linked by a communications medium, message passing is a more common approach.
- In message passing the processes which comprise a concurrent program are linked by *channels*.
- If the two interacting processes are located on the same processor, then this channel could simply be the processor’s local memory.
- If the 2 interacting processes are allocated to different processors, then channel between them is mapped to a physical communications medium between the corresponding 2 processors.
Message Passing Constructs

• There are 2 basic message passing primitives, send & receive
  - **send** primitive: sends a message (data) on a specified channel from one process to another,
  - **receive** primitive: receives a message on a specified channel from other processes.
• The send primitive has different semantics depending on whether the message passing is **synchronous** or **asynchronous**.
• Message passing can be viewed as extending semaphores to convey data as well as synchronisation.

Asynchronous v Synchronous Communication

• **Asynchronous**: (non-blocking)
  - Sender resumes execution as soon as the message is passed to the communication/middleware software

• **Synchronous**: sender is blocked until
  - The OS or middleware notifies acceptance of the message, or
  - The message has been delivered to the receiver, or
  - The receiver processes it & returns a response
Synchronous Message Passing

• In synchronous message passing each channel forms a direct link between two processes.
• Suppose process A is sending data to process B:
  When process A executes send primitive it waits/blocks until process B executes its receive primitive.
• Before the data can be transmitted both A & B must ready to participate in the exchange.
• Similarly the receive primitive in one process will block until the send primitive in the other process has been executed.

Asynchronous Message Passing

• In asynchronous message passing receive has the same meaning/behaviour as in synchronous message passing.
• The send primitive has different semantics.
• This time the channel between processes A & B isn’t a direct link but a message queue.
• Therefore when A sends a message to B, it is appended to the message queue associated with the asynchronous channel, and A continues.
• To receive a message from the channel, B executes a receive removing the message at the head of the channel's message queue and continues.
• If there is no message in the channel the receive primitive blocks until some process adds a message to the channel.
Additions to Asynchronous Message Passing

• Firstly, some systems implement an `empty` primitive which tests if a channel has any messages and returns true if there are no messages.
• This is used to prevent blocking on a receive primitive when there is other useful work to be done in the absence of messages on a channel.
• Secondly, most asynchronous message passing systems implement buffered message passing where the message queue has a fixed length.
• In these systems the `send` primitive blocks on writing to a full channel.

Types of Processes in Message Passing Programs

• **Filters:**
  – These are data transforming processes.
  – They receive streams of data from their input channels, perform some calculation on the data streams, and send the results to their output channels.

• **Clients:**
  – These are triggering processes.
  – They make requests from server processes and trigger reactions from servers.
  – The clients initiate activity, at the time of their choosing, and often delay until the request has been serviced.
Types of Processes in Message Passing Programs (cont’d)

- **Servers:**
  - These are reactive processes.
  - They wait until requests are made, and then react to the request.
  - The specific action taken depends on the request, the parameters of the request and the state of the server.
  - The server may respond immediately or it may have to save the request and respond later.
  - Server is a non-terminating process often servicing more than one client.

- **Peers:**
  - These are identical processes that interact to provide a service or solve a problem.

Message Passing Example 1: Synchronous Network of Filters: Sieve of Eratosthenes

- This method finds primes with each prime sieving for its multiples from the number stream following it.

- The trick is to set up a pipeline of filter processes, of which each one will catch a different prime number.
Ex 1: A Synchronous Network of Filters: Sieve of Eratosthenes (/2)

• Pseudocode

```plaintext
Channel sieve[1](x:int)
process p(1)
  // send out all odd numbers
  for (int i := 3 to N, 2) {send sieve [1] (i)}
end

process p(int i:= 2 to L)
  int pn, num
  receive sieve [i-1] (pn)
  while (1) {
    receive sieve [i-1] (num)
    // pass on num if not a multiple of pn – may be prime
    if (((num mod pn) != 0) { send sieve [i] (num)}
  } // kick off another process
end
```

Asynchronous Heartbeat Algorithms

• Heartbeat algorithms are a typical type of process interaction between peer processes connected together by channels.
• They are called heartbeat algorithms because the actions of each process is similar to that of a heart;
  – first expanding, sending information out;
  – then contracting, gathering new information in.
• This behaviour is repeated for several iterations.
• An example of an asynchronous heartbeat algorithm is the algorithm for computing the topology of a network.
Ex 2: Asynchronous Heartbeat Algorithm for Computing Network Topology

- Each node has a processor and initially only knows about the other nodes to which it is directly connected.
- Algorithm goal is for each node to determine the overall network topology.
- The two phases of the heartbeat algorithm are:
  1. transmit current knowledge of network to all neighbours, and
  2. receive the neighbours’ knowledge of the network.
- After the first iteration the node will know about all the nodes connected to its neighbours, that is within two links of itself.
- After the next it has transmitted (to neighbours) all the nodes with 2 links of itself; and received info about all nodes with 2 links of its neighbours, (i.e. within 3 links of itself).
- In general, after \( i \) iterations knows about all nodes within \((i + 1)\) links of itself.

Firstly from Node 1’s Point of View

....and Node 1 is done!

Next from Node 8’s Point of View

....and Node 8 is done!
Ex 2: Asynchronous Heartbeat Algorithm for Network Topology (/2)

• How many iterations are necessary?
• Since the network is connected, every node has at least one neighbour.
• If known network topology at any given stage is stored in an $n \times n$ matrix $\text{top}$ where
  \[\text{top}[i, j] = \text{true} \text{ if a link exists between node } i \text{ and } j,\]
  then a node knows about the complete topology of the network when every row in top has at least one true value.
• At this point the node needs to perform one more iteration of the heartbeat algorithm to transmit any new information received from one neighbour to its other neighbours.

Ex 2: Asynchronous Heartbeat Algorithm for Network Topology (/3)

• If $m$ is the maximum number of neighbours any node has, and $D$ is the n/w diameter\(^1\), then the number of messages exchanged must be less than $2n \times m \times (D + 1)$.
• A centralised algorithm, in which top was held in memory shared by each process, requires only $2n$ messages. If $m$ and $D$ are small relative to $n$ then there is relatively few extra messages.
• In addition, these messages must be served sequentially by the centralised server. The heartbeat algorithm requires more messages, but these can be exchanged in parallel.

---

\(^1\) i.e. the max. value of the minimum number of links between any two nodes
Ex 2: Asynchronous Heartbeat Algorithm for Network Topology

- All heartbeat algorithms have the same basic structure: send messages to neighbours, and then receive messages from them.
- A major difference between the different algorithms is termination. If the termination condition can be determined locally, as above, then each process can terminate itself.
- If however, termination condition depends on a global condition, each process must iterate a worst-case number of iterations, or communicate with a central controller monitoring the global state of the algorithm, which issues a termination message to each process when required.

Ex 3: Synchronous Heartbeat Algorithm: Parallel Sorting

- To sort an array of n values in parallel using a synchronous heartbeat algorithm, we need to partition the n value equally among the processes.
- Assume that we have 2 processes, $P_1$ and $P_2$, and that n is even.
- Each process initially has $n/2$ values and sorts these values into non descending order, using a sequential sort algorithm.
- Then at each iteration $P_1$ exchanges its largest value with $P_2$’s smallest, and both processes place new values into correct place in their own sorted list of numbers.
- Note: as both sending & receiving block in synchronous message passing, $P_1$ and $P_2$ can’t execute send, receive primitives in same order (as could in asynchronous case).
Ex 3: Synchronous Heartbeat Algorithm: Parallel Sorting: Algorithm Operation (/2)

- Demonstration of Odd/Even Sort for 2 Processes:

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
```

Each Sort Own Array

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```

Ex 3: Synchronous Heartbeat Algorithm: Parallel Sorting: (/3)

- Can extend this to $k$ processes by initially dividing the array to give each process $n/k$ values that it sorts using a sequential algorithm.
- Then we can sort the $n$ elements by repeated applications of the two process compare and exchange algorithm.
- On odd-numbered applications:
  - Every odd-numbered process acts as $P_1$, and every even numbered process acts as $P_2$.
  - Each odd numbered process $P_i$ exchanges data with process $P_{i+1}$.
  - If $k$ is odd, then $P_k$ does nothing on odd numbered applications.
- On even-numbered applications:
  - Even-numbered processes act as $P_1$, odd numbered processes act as $P_2$.
  - $P_1$ does nothing, and $P_k$ does nothing, even if $k$ is even.
The algorithm for odd/even exchange sort on $n$ processes can be terminated in many ways; two of which are:

1. Have a separate controller process who is informed by each process, each round, if they have modified their $n/k$ values.
   - If no process has modified its list then the central controller replies with a message to terminate.
   - This adds an extra $2k$ messages overhead per round.

2. Execute enough iterations to guarantee that the list will be sorted. For this algorithm it requires $k$ iterations.
Ex. 3: Process Odd/Even Exchange Sort in Java

```java
public class OddEvenSort {
    public static void main(String a[]){
        int i;
        int array[] = {12,9,4,99,120,1,3,10};
        odd_even(array, array.length);
    }

    public static void odd_even(int array[], int n){
        /* 1st evens: all these can happen in parallel */
        for (int j = 0; j+1 < n; j += 2)
            if (array[j] > array[j+1]) {
                int T = array[j];
                array[j] = array[j+1];
                array[j+1] = T;
            }
        /* Now odds: all these can happen in parallel */
        for (int j = 1; j+1 < array.length; j += 2)
            if (array[j] > array[j+1]) {
                int T = array[j];
                array[j] = array[j+1];
                array[j+1] = T;
            }
    }
}
```

SECTION 5.2: PARALLEL PROGRAMMING WITH MESSAGE PASSING INTERFACE & OPENMP
When is Parallel Implementation Useful?

• **In general it is useful for large problems**
  
  – i.e. you know what speed-up to expect,
  – But you need to be able to recognise them!

• Three types of problems are suitable:
  
  1. **(Embarasingly) Parallel Problems**
  2. **Regular and Synchronous Problems**
  3. **Irregular and/or Asynchronous Problems**

1. **(Embarasingly) Parallel problems:**
   
   – The problem can be broken down into subparts with each independent of the others
   – No comms required, except to split up problem/combine results
   – Linear speed-up can be expected
   – Example of this is: Monte-Carlo simulations

2. **Regular and Synchronous Problems:**
   
   – Same instruction set (regular algorithm) applied to all data
   • (**) Synchronous comms: each processor finishes its task at the same time
   • Local (neighbour to neighbour) & collective (combine final results) comms
   – Computation: comms ratio dictates speed-up
   • If large, good speed-up for local comms & ok speed-up for non-local comms
   – Ex: matrix-vector products, sorting (loosely synch.)

3. **Irregular and/or Asynchronous Problems:**
   
   – Irregular algorithm which cannot be implemented efficiently except with message passing and high communication overhead
   – Comms usually asynchronous - needs care in load balancing
   • Often dynamic data repartitioning between processors required
   • Speed-up hard to predict; easier to split into regular/irregular parts
   – Ex: Melting ice problem (or any moving boundary simulation)
Example 1: matrix-vector product

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix}
\times
\begin{pmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4
\end{pmatrix}
= \begin{pmatrix}
c_1 \\
c_2 \\
c_3 \\
c_4
\end{pmatrix}
\]

with

\[
\begin{align*}
c_1 &= a_{11} \times b_1 + a_{12} \times b_2 + a_{13} \times b_3 + a_{14} \times b_4 \\
c_2 &= a_{21} \times b_1 + a_{22} \times b_2 + a_{23} \times b_3 + a_{24} \times b_4 \\
c_3 &= a_{31} \times b_1 + a_{32} \times b_2 + a_{33} \times b_3 + a_{34} \times b_4 \\
c_4 &= a_{41} \times b_1 + a_{42} \times b_2 + a_{43} \times b_3 + a_{44} \times b_4
\end{align*}
\]

• A parallel approach:
  • Each element of vector \( c \) depends on vector \( b \) and only one line of \( A \)
  • So each \( c \) can be calculated independently from the others
  • Communication only needed to split problem & combine final results
• => a linear speed-up can be expected for large matrices

Example 2: Monte-Carlo calculation of Pi

• Monte Carlo Integration
  • \( \pi = 3.14159…. = \) area of a circle of radius 1
  • \( \pi/4 = \) fraction of points in circle quadrant
  • More points => more accurate value for \( \pi \)
• A parallel approach:
  – Each point is randomly placed within the square & so each point’s position is independent of the position of the others
  – Can split problem by letting each node randomly place a given number of points
  – Only need communicate number of points & take in final results
• => Can expect linear speed-up allowing for a larger number of points and hence greater accuracy in the estimation of \( \pi \).
Example 3: A More Real Problem

- **Knight’s Tour Problem**
- After each move, chess s/w must find best move
  - This set is large, but finite
- Each move from this set can be evaluated independently & the set can be partitioned
- Comms only needed to split problem and combine the final results
  - A linear speed-up can be expected
  - This means that, in a reasonable time, moves can be studied more thoroughly
  - This depth of evaluation makes s/w more competitive

**SECTION 5.2.1: MESSAGE PASSING INTERFACE (MPI)**
Some background on MPI

- **MPI**
  - Developed by MPI forum (Industry, Academia & Govt.)
  - 1994: Set up a standardised Message-Passing Interface (MPI-1)
  - It was intended as an interface to both C and FORTRAN.
  - Aim was to provide a specification implementable on any parallel computer or cluster => portability of code was a big aim
  - BUT Implementation on Shared Memory Architectures often poor

- MPI provides support for:
  - Point-to-point & collective (i.e. group) communications
  - Inquiry routines to query the environment (how many nodes are there, which node number am I, etc.)
  - Constants and data-types

- We will start with the basics: initialising MPI, and using point-to-point communication

### MPI Preliminaries...

- **Naming convention**
  - All MPI identifiers are prefixed by ‘MPI_’.
  - C routines contain lower case (i.e. ‘MPI_Init’),
  - Constants are all in upper case (e.g. ‘MPI_FLOAT’ is an MPI C data-type).
  - C routines are actually integer functions which return a status code (you are strongly advised to check these for errors!).

- **Running MPI**
  - Number of processors used is specified in the command line, when running the MPI loader that loads the MPI program onto the processors, to avoid hard-coding this into the program
  - e.g. `mpirun -np N exec`
MPI Preliminaries... (/2)

- Writing a program using MPI: what is parallel, what is not
  - Only one program is written. By default, every line of the code is executed by each node running the program.
  - E.g. if code contains `int result=0`, each node will locally create a variable and assign the value.
- When a section of the code needs to be executed by only a subset of nodes, it has to be explicitly specified.
- E.g., providing that we are using 8 nodes, and that MyID stores the rank of the node (from 0 to 7), this section of code assigns to result zero for the first half of them, and unity for the second.

```c
int result;
if(MyID < 4) result = 0;
else result = 1;
```

Common MPI Routines

- MPI has a ‘kitchen sink’ approach of 129 different routines
- Most basic programs can get away with using six.
- As usual use `#include "mpi.h"` in C.

```c
MPI_Init Initialise MPI computation
MPI_Finalize Terminate MPI computation
MPI_Comm_size Determine number of processes
MPI_Comm_rank Determine my process number
MPI_Send, MPI_Isend Blocking, non-blocking send
MPI_Recv, MPI_Irecv Blocking, non-blocking
```
Common MPI Routines (/2):
MPI Initialisation, Finalization

- In all MPI programs, must initialise MPI before use & finalise at end
- All MPI-related commands and types must be handled within this section of code:

```
MPI_Init
  ...
MPI_Finalize
```

- **MPI_Init** takes two parameters as input (argc and argv),
  - It is used to start the MPI environment, create the default communicator (more later) and assign a rank to each node.
- **MPI_Finalize** cleans up all MPI state. Once this routine is called, no MPI routine (even **MPI_INIT**) may be called.
- The user must ensure that all pending communications involving a process completes before the process calls **MPI_Finalize**.

Common MPI Routines (/3):
Basic Inquiry Routines

- At various stages in a parallel-implemented function, useful to know how many nodes program is using, or what current node’ rank is.
- The **MPI_Comm_size** function returns the number of processes/nodes as an integer, taking only one parameter, a communicator.
- In most cases you will only use the default Communicator: **MPI_COMM_WORLD**.
- The **MPI_Comm_rank** function is used to determine what the rank of the current process/node on a particular communicator.
  - E.g. if there are two communicators, it is possible, and quite usual, that the ranks of the same node would differ.
  - Again, in most cases, this function will only be used with the default communicator as an input (**MPI_COMM_WORLD**), and it will return (as an integer) the rank of the node on that communicator.
Common MPI Routines (/4):
Point-to-Point communications in MPI

• This involves communication between two processors, one sending, and the other receiving.
• Certain information is required to specify the message:
  – Identification of sender processor
  – Identification of destination/receiving processor
  – Type of data (MPI_INT, MPI_FLOAT etc)
  – Number of data elements to send (i.e. array/vector info)
  – Where the data to be sent is in memory (pointer)
  – Where the received data should be stored in (pointer)

Common MPI Routines (/5):
Sending data MPI_Send, MPI_Isend

• MPI_Send is used to perform a blocking send, (i.e. process waits for the communication to finish before going to the next command).

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

• This function takes six parameters:
  – the location of the data to be sent i.e. a pointer (input parameter)
  – the number of data elements to be sent (input parameter)
  – the type of data e.g. MPI_INT, MPI_FLOAT, etc. (input parameter)
  – the rank of the receiving/destination node (input parameter)
  – a tag for identification of the communication (input parameter)
  – the communicator to be used for transmission (input parameter)

• MPI_Isend is non-blocking, so an additional parameter, to allow for verification of communication success is needed.
• It is a pointer to an element of type MPI_Request.
Common MPI Routines (/6):
Receiving data MPI_Recv, MPI_Irecv

- **MPI_Recv** is used to perform a blocking receive, (i.e. process waits for the communication to finish before going to the next command).

```c
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- This function takes seven parameters:
  - the location of the receive buffer i.e. a pointer (*output parameter*)
  - the max number of data elements to be received (*input parameter*)
  - the type of data e.g. MPI_INT, MPI_FLOAT, etc. (*input parameter*)
  - the rank of the source/sending node (*input parameter*)
  - a tag for identification of the communication (*input parameter*)
  - the communicator to be used for transmission (*input parameter*)
  - a pointer to a structure of type MPI_Status, contains source processor’s rank, communication tag, and error status (*output parameter*)

- For the non-blocking **MPI_Irecv**, **MPI_Request** replaces **MPI_Status**.

A first MPI example: Hello World.

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
    int myid, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("process %d out of %d says Hello\n", myid, size);
    MPI_Finalize();
    return 0;
}
```
Example 2: Exchanging 2 Values

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, value, size, length = 1, tag = 1;

    MPI_Status status;

    /* Initialize MPI and get own id (rank) */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size!=2) {
        printf("use exactly two processes\n");
        exit(1);
    }

    if (myid == 0) {
        otherid = 1; myvalue = 14;
    } else {
        otherid = 0; myvalue = 25;
    }

    printf("process %d sending %d to process %d\n", myid, myvalue, otherid);
    /* Send one integer to the other node (i.e. "otherid") */
    MPI_Send(&myvalue, 1, MPI_INT, otherid, tag, MPI_COMM_WORLD);

    /* Receive one integer from any other node */
    MPI_Recv(&othervalue, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    printf("process %d received a %d\n", myid, othervalue);

    MPI_Finalize(); /* Terminate MPI */

    return 0;
}
```

Compiling and Running MPI Programs

- To compile programs using MPI, you need an “MPI-enabled” compiler.
- On cluster, can use `mpicc` to compile C programs containing MPI commands or `mpicxx` for C++.
- Before running an executable using MPI, you need to make sure the "multiprocessing daemon" (MPD) is running.
- It makes the workstations into "virtual machines" to run MPI programs.
- When you run an MPI program, requests are sent to MPD daemons to start up copies of the program.
- Each copy then uses MPI to communicate with other copies of the same program running in the VM. Just type “mpd &” in the terminal.
- To run the executable, type “`mpirun -np N ./executable_file`”, where N is the number to be used to run the program.
- This value is then used in your program by `MPI_Init` to allocate the nodes and create the default communicator.
Example 3: “Ring” Communication

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
  int rank, value, size;
  MPI_Status status;
  /* Initialize MPI and get own id (rank) */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  do {
    if (rank == 0) {
      /* Master Node sends the value */
      MPI_Send(&value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
    } else {
      /* Slave Nodes block on receive the send on the value */
      MPI_Recv(&value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);
      if (rank < size - 1) {
        MPI_Send(&value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
      }
      printf("process %d got %d\n", rank, value);
    } while (value > 0);
    /* Terminate MPI */
    MPI_Finalize();
    return 0;
  }
```

Example 4: Matrix-Vector Product Implementation

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
  int A[4][4], b[4], c[4], line[4], temp[4], local_value, myid;
  MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == 0) {
    for (int i = 0; i <= 4; i++) {
      for (int j = 0; j <= 3; j++)
        A[i][j] = i * j; /* set some notional values for A, b */
    }
  } else {
    MPI_Recv(line, 4, MPI_INT, 0, myid, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (int i = 1; i <= 4; i++) {
      if (myid == 0) {
        MPI_Send(&c[myid], 1, MPI_INT, 0, myid, MPI_COMM_WORLD);
      }
    }
  }
  MPI_Finalize();
  return 0;
```

Example 5: Pi Calculation Implementation

```c
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (myid == 0) {
        for (int i=1; i<size; i++) {
            MPI_Send(&points, 1, MPI_INT, i, i, MPI_COMM_WORLD);
        }
    } else
        MPI_Recv(&points, 1, MPI_INT, 0, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    rands = new double[2*points];
    for (int i=0; i<2*points; i++)
        rands[i]=random();
    for (int i=0; i<points; i++) {
        x = rands[2*i]/INT_MAX;
        y = rands[2*i+1]/INT_MAX;
        if (x*x+y*y<1) inside++ /* point is inside unit circle so incr var inside */
    }
    delete[] rands;
    if (myid == 0) {
        for (int i=1; i<size; i++) {
            int temp; /* master receives all inside values from slaves */
            MPI_Recv(&temp, 1, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            inside+=temp; /* master sums all insides sent to it by slaves */
        }
        MPI_Send(&inside, 1, MPI_INT, 0, i, MPI_COMM_WORLD); /* send inside to master*/
        if (myid == 0) {
            Pi_comp = 4*double(inside) / double(size*points);
            cout << "Value obtained: " << Pi_comp << endl << "Pi: " << Pi_real << endl;
        }
    }
    MPI_Finalize(); return 0;
}
```

Collective communications in MPI

- Groups are sets of processors interacting with each other in a certain way.
- Such communications permit a more flexible mapping of the language to the problem (allocation of nodes to subparts of the problem etc).
- MPI implements Groups using data objects called Communicators.
- A special Communicator is defined (called ‘MPI_COMM_WORLD’) for the group of all processes.
- Each Group member is identified by a number (its Rank 0..n-1).
- There are three steps to create new communication structures:
  - accessing the group corresponding to MPI_COMM_WORLD,
  - using this group to create sub-groups,
  - allocating new communicators for this group.
- We will see this in more detail in the last examples.
Some Sophisticated MPI Routines

- Advantage of global comms routines below is MPI system implements them more efficiently than the coder, with fewer function calls.
- Also the system will have more opportunity to overlap message transfers with internal processing and to exploit parallelism that might be available in the communications network.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Barrier</td>
<td>Synchronise</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>Broadcast same data to all procs</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>Get data from all procs</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>Send different data to all procs</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>Combine data from all onto one proc</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>Combine data from all procs onto all procs</td>
</tr>
</tbody>
</table>

Sophisticated MPI Routines:

**MPI_Barrier**

- **MPI_Barrier** is used to synchronise a set of nodes.
  ```c
  int MPI_Barrier( MPI_Comm comm )
  ```
- It blocks the caller until all group members have called it.
- ie call returns at any process only after all group members have entered the call.
- This functions takes only parameter, the communicator (i.e. group of nodes) to be synchronised.
- As we previously saw with other functions, it will most of the times be used with the default communicator, `MPI_COMM_WORLD`.
Sophisticated MPI Routines: **MPI_Bcast**

- **MPI_Bcast** used to send data from one node to all the others in one single command.

```c
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )
```

- This function takes five parameters:
  - location of data to be sent i.e. a pointer (*input/output* parameter)
  - number of data elements to be sent (*input* parameter)
  - type of data (*input* parameter)
  - rank of the broadcast node (*input* parameter)
  - communicator to be used (*input* parameter)

![Diagram of MPI_Bcast]

---

Sophisticated MPI Routines: **MPI_Gather**

- **MPI_Gather** used to gather on 1 node data scattered over a group.

```c
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- This function takes eight parameters:
  - location of data to be sent i.e. a pointer (*input* parameter)
  - number of data elements to be sent (*input* parameter)
  - type of data to be sent (*input* parameter)
  - location of the receive buffer i.e. a pointer (*output* parameter)
  - number of elements to be received (*input* parameter)
  - type of data to be received (*input* parameter)
  - rank of the sending node (*input* parameter)
  - communicator to be used for transmission. (*input* parameter)

![Diagram of MPI_Gather]
Sophisticated MPI Routines: **MPI_Scatter**

* MPI **Scatter** used to scatter data from single node to a group

```c
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recvnt, MPI_Datatype recvtype, int root,
                MPI_Comm comm)
```

- This function takes eight parameters:
  - location of data to be sent i.e. a pointer *(input parameter)*
  - number of data elements to be sent *(input parameter)*
  - type of data to be sent *(input parameter)*
  - location of the receive buffer i.e. a pointer *(output parameter)*
  - number of elements to be received *(input parameter)*
  - type of data to be received *(input parameter)*
  - rank of the sending node *(input parameter)*
  - communicator to be used for transmission. *(input parameter)*

![MPI Scatter Diagram]

Sophisticated MPI Routines: **MPI_Reduce**

* MPI **Reduce** used to reduce values on all nodes of a group to a single value on one node using some reduction operation (sum etc).

```c
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )
```

- This function takes seven parameters:
  - location of the data to be sent i.e. a pointer *(input parameter)*
  - location of the receive buffer i.e. a pointer *(output parameter)*
  - number of elements to be sent *(input parameter)*
  - type of data e.g. MPI_INT, MPI_FLOAT, etc. *(input parameter)*
  - operation to combine the results e.g. MPI_SUM *(input parameter)*
  - identity of root node *(input parameter)*
  - communicator used for transmission *(input parameter)*

![MPI Reduce Diagram]
Sophisticated MPI Routines: MPI_Allreduce

- **MPI_Allreduce** is used to reduce values on all group nodes to a one value, and send it back to all (i.e. equals MPI_Reduce+MPI_Bcast)

```c
int MPI_Allreduce ( void *sendbuf, void *recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

- This function takes six parameters:
  - location of the data to be sent i.e. a pointer (input parameter)
  - location of the receive buffer i.e. a pointer (output parameter)
  - number of elements to be sent (input parameter)
  - type of data e.g. MPI_INT, MPI_FLOAT, etc. (input parameter)
  - operation to combine the results e.g. MPI_SUM (input parameter)
  - communicator used for transmission (input parameter)

Example 6: A New Matrix-Vector Product

```c
MPI_Bcast(b, 4, MPI_INT, 0, MPI_COMM_WORLD); if (myid == 0) {
    for (int i=0; i<4; i++) { /* slaves do most multiplying */
        MPI_Send( temp, 4, MPI_INT, i, i, MPI_COMM_WORLD); /* No need to send vector b here */
    }
} else {
    MPI_Recv( line, 4, MPI_INT, 0, myid, MPI_COMM_WORLD, MPI_STATUS_IGNORE); /* No need to receive vector b here */
} /* master node does its share of multiplication too*/
    for (int i=1; i<4; i++) {
        MPI_Recv( &c[i], 1, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
MPI_Finalize(); return 0;
```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    #define INT_MAX_1000000000
    int myid, size, inside=0, outside=0, points=10000;
    double x, y, Pi_comp, Pi_real=3.141592653589793238462643;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* Again send/receive replaced by MPI_Bcast */
    MPI_Bcast(&points,1,MPI_INT, 0, MPI_COMM_WORLD);
    srand(time(NULL));
    for (int i=0; i<points; i++) {
        x = rand()/INT_MAX_;
        y = rand()/INT_MAX_;
        if((x*x+y*y)<1) inside++ /* point is inside unit circle so incr var inside */
    }
    delete[] rands;
    if (myid == 0) {
        for (int i=1; i<size; i++) {
            MPI_Recv(&temp, 1, MPI_INT, i, i,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            inside+=temp; /* master gets all inside values from slaves */
        }
        MPI_Reduce(&inside, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        if (myid == 0) {
            count = 4 * (double)inside / (double)(size*points);
            cout << "Value obtained: " << Pi_comp << endl << "Pi: " << Pi_real << endl;
        }
        MPI_Finalize(); return 0;
    }
}

Example 7: A New Pi Calculation Implementation

Using Communicators

• Creating a new group (and communicator) by excluding the first node:

```
#include <mpi.h>
int main(int argc, char *argv[]) {
    ...
    MPI_Comm comm_world, comm_worker;
    MPI_Group group_world, group_worker;
    comm_world = MPI_COMM_WORLD;
    MPI_Comm_group(comm_world, &group_world);
    MPI_Group_excl(group_world, 1, 0, &group_worker);
    MPI_Comm_create(comm_world, group_worker, &comm_worker);
    ...
}
```

• Warning:

`MPI_Comm_create()` is a collective operation, so all processes in the old communicator must call it - even those not going in the new communicator.

Lecture 5: Message-Oriented Communication  CA4006 Lecture Notes (Martin Crane 2015)  57
Example 8: Using Communicators

```c
#include <mpi.h>
#include <stdio.h>
#define NPROCS 8

int main(int argc, char *argv[]) {
    int rank, newrank, sendbuf, recvbuf;
    ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
    MPI_Group orig_group, new_group;
    MPI_Comm new_comm

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    sendbuf = rank;
    /* Extract the original group handle */
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
    if (rank < NPROCS/2) { /* Split tasks into 2 distinct groups based on rank */
        MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
    } else
        MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
    /* Create new communicator and then perform collective communications */
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
    MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
    MPI_Group_rank (new_group, &new_rank);

    printf("rank= %d newrank= %d recvbuf= %d\n", rank, newrank, recvbuf);
    MPI_Finalize();
}
```

SECTION 5.2.2: OPEN MULTIPROCESSING (OPENMP)
Introduction to OpenMP

• Stands for *Open Multi-Processing*, or *Open specifications for Multi-Processing*

• Represents collaboration between interested parties from h/w and s/w industry, government and academia.

• An API to facilitate explicitly direct multi-threaded, shared memory parallelism.

• Supported in C, C++, and Fortran, and on most processor architectures and OS.

• It consists of a set of compiler directives, library routines, and environment variables affecting run-time behaviour.

• Introduce it here as complementary to and usable in conjunction with MPI to achieve speedup

Motivations to use OpenMP

• Provides a standard among a variety of shared memory architectures/platforms.
  – Currently at OpenMP Version 4.0 (as of July 2013)
  – More details at openmp.org/wp/resources/

• Establishes a simple and limited set of directives for programming shared memory machines.
  – As with MPI, can get quite good parallelism using 3 or 4 directives.

• Unlike MPI:
  – Facilitates incremental parallelization of a serial program,
  – Does not require ‘all or nothing’ approach to parallelization,
  – MPI scales well but is non-trivial to implement for codes originally written for serial machines & not good for shared memory

• Can implement both coarse-grain & fine-grain parallelism.
What OpenMP is not

• OpenMP Is Not:
  – Meant for distributed memory parallel systems (by itself)
  – Necessarily implemented identically by all vendors
  – Guaranteed to make the most efficient use of shared memory

• NB OpenMP will not:
  – Check for data dependencies, data conflicts, race conditions, or deadlocks
  – Check for code lines that cause program to be classified as non-conforming
  – Cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization

OpenMP Programming Model

• Shared Memory Model:
  – OpenMP is designed for multi-processor/core, shared memory machines.
  – The underlying architecture can be shared memory UMA or NUMA.
OpenMP Programming Model (/2)

– The Fork-Join Model

Data Dependencies
• Data on one thread can be dependent on data on another one
• This can result in wrong answers
  – Thread 0 may require a variable that is calculated on thread 1
  – Answer depends on timing – When thread 0 does the calculation, has thread 1 calculated it’s value yet?
• Example – Fibonacci Sequence 0, 1, 1, 2, 3, 5, 8, 13, ... more bunnies!

```c
A[1] = 0;
for(i = 3; i <= 100; i++){
}
```

• Parallelize on 2 threads
  – Thread 0 gets i = 3 to 51, Thread 1 gets i = 52 to 100
    • Look carefully at calculation for i = 52 on thread 1
    • What will be values of for i – 1 and i – 2?
Data Dependencies (/2)

• **A Test for Dependency:**
  – If serial loop is executed in reverse order, will it give same result?
  – If so, it’s ok
  – You can test this on your serial code

• What about subprogram calls?
  ```
  for(i = 0; i < 100; i++){
    mycalc(i,x,y);
  }
  ```

• Does the subprogram write x or y to memory?
  – If so, they need to be private
  – Variables local to subprogram are local to each thread

• Be careful with global variables and common blocks

Parallelism in OpenMP

• **Thread-Based Parallelism**
  – OpenMP programs accomplish parallelism solely using *threads*.
  – A *thread of execution* is the smallest unit of processing that can be scheduled by an operating system.
    • Analogous conceptually to a subroutine that can be scheduled to run autonomously.
  – These threads exist within the resources of a single process, without which they cannot exist.
  – Usually number of threads match the number of machine processors/cores.
    • However, the actual use of threads is up to the application.
Parallelism in OpenMP (/2)

• **Explicit Parallelism**
  
  – OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
  
  – Parallelization can be as simple as taking a serial program and inserting compiler directives....
  
  – The general form of these are:
    
    ```c
    #pragma omp construct [clause [clause]...]
    ```
    
    • Example of this:
    
    ```c
    #pragma omp parallel num_threads(4)
    ```
  
  – **Note about #pragma**
    
    • These are special preprocessor instructions.
    
    • Typically added to system to allow behaviours that aren’t part of the basic language specification.
    
    • Compilers that don’t support the pragmas ignore them.

---

Example 1: My first OpenMP Code.

```c
#include <stdio.h>
int main(void) {
    #pragma omp parallel num_threads(2)
    printf("Hello, world.\n");
    return 0;
}
```

• **Thread Creation**
  
  – `pragma omp parallel` used to fork additional threads (here 2) to carry out the work enclosed in the construct in parallel.
  
  – The original thread is denoted as master thread with thread ID 0.
  
  – `num_threads(2)` is one of a number of clauses that can be specified e.g. `private` variables, `shared` variables, `reduction` operation
  
  – Simple Example: Display "Hello, world." using multiple threads.
  
  – Complex: insert subroutines to set multiple levels of parallelism, locks and even nested locks.
Example 1: My first OpenMP Code (/2).

- **Thread Creation (/2)**
  - When a thread reaches a `parallel` directive, it creates a team of threads and becomes the master of the team.
  - Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
  - There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
  - If any thread terminates within a parallel region, all threads in the team will terminate.
  - If this happens, the work done up until that point is undefined.

Running this Example in OpenMP

- Use flag `-fopenmp` to compile using GCC:
  
  ```
  $ gcc -fopenmp hello.c -o hello
  ```

- Outputs on a computer with 2 cores, and thus 2 threads:
  ```
  Hello, world.
  Hello, world.
  ```

- However, output may also be garbled due to race condition caused from the two threads sharing the standard output:
  ```
  Hello, wHello, wworld.
  rld.
  ```

- A helpful step by step example on how to run can be found at dartmouth.edu/~rc/classes/intro_openmp/compile_run.html
Example 2: More Complex OpenMP Code.

```c
#include <stdio.h>
int main(int argc, char **argv) {
    int a[100];
    #pragma omp parallel for
    int i;
    for (i = 0; i < 100; i++)
        a[i] = 2 * i;
    return 0;
}
```

- **Work-sharing constructs**
  - `omp for`/`omp do` for forking extra threads to do work enclosed in `parallel` (aka `loop` constructs).
  - This is equivalent to:
    ```c
    { // stuff here
    #pragma omp parallel //nb omp twice
    int i;
    #pragma omp for (i = 0; i < 100; i++)
    a[i] = 2 * i;
    return 0;
    }
    ```

Other Work Constructs in OpenMP

- **sections**
  Used to assign consecutive but independent code blocks to different threads

- **single**
  Specifying a code block that is executed by only one thread, a barrier is implied in the end
  Uses first thread that encounters the construct.

- **master**
  Similar to single, but the code block will be executed by the `master` thread only (thread executing outside the parallel regions)
  No barrier implied in the end.
OpenMP Work Constructs (Summary)

- **do/for** - shares loop iterations across team. Akin to "data parallelism"
- **sections** - breaks work into separate, discrete sections. Each executed by a thread. Can be used to implement a type of "functional parallelism".
- **single** - serializes a chunk of code

Example 3: **Sections** Construct.

```c
void XAXIS(); void YAXIS(); void ZAXIS();
void sect_example()
{
#pragma omp parallel sections
{
#pragma omp section
  XAXIS();
#pragma omp section
  YAXIS();
#pragma omp section
  ZAXIS();
}
```

- **Purpose**
  - This **sections** directive is used to execute routines XAXIS, YAXIS, and ZAXIS concurrently
Example 4: single Construct.

```c
#include <stdio.h>
void work1() {}
void work2() {}
void single_example() {
#pragma omp parallel
{
#pragma omp single
    printf("Beginning work1.\n");
    work1();
#pragma omp single
    printf("Finishing work1.\n");
#pragma omp single nowait
    printf("Finished work1, starting work2.\n");
    work2();
}
```

- **Purpose**
  - The `single` directive specifies that the enclosed code is to be executed by only one thread in the team.
  - Useful dealing with sections of code that are not thread safe (such as I/O).
  - There is an implicit barrier at end of each except where a `nowait` clause is specified.

Synchronisation Constructs in OpenMP

- **master**
  - Strictly speaking, `master` is a synchronisation directive - master thread only and no barrier implied in the end.

- **critical**
  - Specifies a critical section i.e. a region of code that must be executed by only one thread at a time.

- **atomic**
  - Commonly used to update counters and other simple variables that are accessed by multiple threads simultaneously.

- **barrier**
  - Synchronizes all threads in the team.
  - When reached, a thread waits there until all other threads have reached that barrier.
  - All then resume executing in parallel the code that follows the barrier.
Example 5: Data Scope Attributes

```c
#include <stdio.h>
int a, b=0;
#pragma omp parallel for private(a) shared(b)
for(a=0; a<50; ++a)
{
    #pragma omp atomic // means that either happens or doesn’t.
    b += a; // one thread can’t interrupt another here
}
```

- **Purpose**
  - These attribute clauses specify data scoping/sharing.
  - As OpenMP based on shared memory programming model, most variables shared by default.
  - Used with directives e.g. `Parallel`, `Do/for`, `Sections` to control the scope of enclosed variables.
  - Here `a` is explicitly specified as `private` (each thread has their own copy of it) and that `b` is `shared` (each thread accesses the same variable).

Example 6: A more Complex HelloWorld

```c
#include <iostream>
using namespace std;
#include <omp.h>
int main(int argc, char *argv[])
{
    int th_id, nthreads;
    #pragma omp parallel private(th_id) shared(nthreads)
    {
        th_id = omp_get_thread_num(); // returns thread id
        #pragma omp critical // only one thread can access this at a time!
        { // only one thread can access this at a time!
            cout << "Hello World from thread " << th_id << \n';
        }
        #pragma omp barrier // one thread waits for all others
        #pragma omp master // master thread access only!
        { // master thread access only!
            nthreads = omp_get_num_threads(); // returns number of thread
            cout << "There are " << nthreads << " threads" << \n';
        }
    }
    return 0;
}
```

- **Purpose**
  - `Private`, `shared` declares that threads have their own copy of the variable or share a copy, respectively.
Reduction Clauses

- **Reduction**
  - Like MPI, OpenMP supports the Reduction operation.

    ```c
    int t;
    #pragma omp parallel reduction(+:t)
    {
        t = omp_get_thread_num() + 1;
        printf("local %d\n", t);
    }
    printf("reduction %d\n", t);
    ```

  - Reduction Operators: + * logical operators and Min(), Max()
  - The operation makes the specified variable private to each thread.
  - At the end of the computation it combines private results
  - Very useful when combined with `for` as shown below see below:

    ```c
    sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for (i=0; i < 100; i++) {
        sum += array[i];
    }
    ```

- The code fragment will be successfully compiled, and the `#pragma omp for` directive will be simply ignored by the compiler.
- So only one thread executes the loop, and it could be tricky for a developer to uncover.
- The correct form should be:

  ```c
  #pragma omp parallel //this is correct
  {
      #pragma omp for
      //your code
  }
  ```

Common Mistakes in OpenMP:

#1 Missing **Parallel** keyword

- The code fragment will be successfully compiled, and the `#pragma omp for` directive will be simply ignored by the compiler.
- So only one thread executes the loop, and it could be tricky for a developer to uncover.
- The correct form should be:
Common Mistakes in OpenMP:

#2 Missing for keyword

- `#pragma omp parallel`

This directive may be applied to a single code line as well as to a code fragment. This may cause unexpected behaviour of the `for` loop:

```c
#pragma omp parallel num_threads(2) // incorrect as for keyword omitted
for (int i = 0; i < 10; i++)
    myFunc();
```

- If the developer wanted to share the loop between two threads, they should use the `#pragma omp parallel for` directive.

- Here the loop would have been executed 10 times indeed.

- However, the code above will be executed once in every thread. As the result, the `myFunc()` function will be called 20 times.

- The correct version of the code is provided below:

```c
#pragma omp parallel for num_threads(2) // now correct
for (int i = 0; i < 10; i++)
    myFunc();
```

Common Mistakes in OpenMP:

#3 Redundant Parallelization

- Applying the `#pragma omp parallel` directive to a large code fragment can lead to unexpected behaviour in cases like below:

```c
#pragma omp parallel num_threads(2)
{
    ... // some lines of code
    #pragma omp parallel for
    for (int i = 0; i < 10; i++)
    {
        myFunc();
    }
}
```

- A naïve programmer wanting to share the loop execution between two threads placed the `parallel` keyword inside a parallel section.

- The result of the code execution will be similar to the previous example: the `myFunc` function will be called 20 times, not 10.

- The correct version of the code is the same as the above except for

```c
#pragma omp parallel for
for (int i = 0; i < 10; i++)
```
Common Mistakes in OpenMP:

#4 Redefining the Number of Threads in a Parallel section

- By OpenMP 2.0 spec the number of threads cannot be redefined inside a parallel section without run-time errors and program termination:

```c
#pragma omp parallel
{
    #pragma omp set_num_threads(2); // incorrect to call this routine here
    #pragma omp for
    for (int i = 0; i < 10; i++)
    
        myFunc();
}
```

- A correct version of the code is:

```c
#pragma omp parallel num_threads(2) // correct!
{
    #pragma omp for
    for (int i = 0; i < 10; i++)
    
        myFunc();
}
```

- Or:

```c
omp_num_threads(2) // also
#pragma omp parallel // correct!
{
    #pragma omp for
    ... // more code
```

SECTION 5.2.3: THE BEST OF BOTH WORLDS... HYBRIDIZATION OF MPI & OPENMP
MPI & OpenMP: Advantages & Disadvantages

Pure MPI Pros:
- Portable to distributed and shared memory machines.
- Scales beyond one node.
- No data placement problem.

Pure MPI Cons:
- Difficult to develop and debug.
- Explicit communication.
- High latency, low bandwidth.
- Coarse granularity.
- Difficult load balancing.

Pure OpenMP Pros:
- Easy to implement parallelism.
- Coarse and fine granularity.
- Implicit Communication.
- Low latency, high bandwidth.
- Dynamic load balancing.

Pure OpenMP Cons:
- Only on shared memory machines.
- No specific thread order.
- Scale within one node.
- Data placement problem possible.

Hybridization: What it is & How it Helps

What it is:
- Using inherently different models of programming in a complimentary manner, to achieve a benefit not possible otherwise.
- A way to use different models of parallelization in a way that takes advantage of the good points of each.
- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures.
- Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes.
- Good usage of shared memory system resource (memory, latency, and bandwidth).
Hybridization: How it Helps

- **Generalities:**
  - Some problems have two-level parallelism naturally.
  - Some problems could only use restricted number of MPI tasks.
  - *Could have* better scalability than both pure MPI and pure OpenMP.

- **How it helps:**
  - Adding MPI to OpenMP code can help scale across multiple SMP nodes;
  - Adding OpenMP to MPI code can use shared memory on SMP nodes more efficiently, and reduce explicit intra-node communication needs;
  - Adding MPI and OpenMP during the design/coding of a new program can help maximize efficiency, performance, and scaling; Avoids the extra communication overhead with MPI within node.
  - OpenMP adds fine granularity (larger message sizes) and allows increased and/or dynamic load balancing.

---

Hybrid MPI + OpenMP programming

- Each MPI process spawns multiple OpenMP threads

```
mpirun

rank0

FORK
JOIN
(parallel region)

rank1

FORK
JOIN
(parallel region)
```

---
Hybrid MPI + OpenMP Example 1:

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

int main(int argc, char *argv[]) {
    int numprocs, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int iam = 0, np = 1;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);
    #pragma omp parallel default(shared) private(iam, np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d from process %d out of %d on%s\n", iam, np, rank, numprocs, processor_name);
    }
    MPI_Finalize();
}
```

Lecture 5: Message-Oriented Communication

Sample Output

Hello from thread 0 out of 4 from process 0 out of 2 on morab006
Hello from thread 1 out of 4 from process 0 out of 2 on morab006
Hello from thread 2 out of 4 from process 0 out of 2 on morab006
Hello from thread 3 out of 4 from process 0 out of 2 on morab006
Hello from thread 0 out of 4 from process 1 out of 2 on morab001
Hello from thread 3 out of 4 from process 1 out of 2 on morab001
Hello from thread 1 out of 4 from process 1 out of 2 on morab001
Hello from thread 2 out of 4 from process 1 out of 2 on morab001
Hybrid MPI + OpenMP

Example 2: Calculate $\pi$

- $\int_0^1 \frac{dx}{1+x^2} = \tan^{-1} 1 = \frac{\pi}{4}$
- Integrating the function $f(x)$ from [0,1] will give approximation to $\pi$
- Each MPI process integrates over a range of width $1/\text{numproc}$, as a discrete sum of $\text{num\_step}$ steps, each of width $\text{step}$
- In each MPI process, $n\text{threads}$ OpenMP threads perform part of the sum in OPENMP alone.

---

Hybrid MPI + OpenMP Example 2 (/2): \texttt{omppi.c} code

```c
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel for reduction(+:sum) private(x)
  for (i = 0; i < num_steps; i++){
    x = (i+0.5)*step; // scales x in terms of step
    sum = sum + 4.0/(1.0+x*x); // sum is private til threads done
  }
  pi = step * sum;
}
```

This is the OpenMP version of the Monte-Carlo calculation on its own
Hybrid MPI + OpenMP Example 2 (/3)

**mpi.c**

```c
#include <mpi.h>

void main (int argc, char *argv[])
{
    int i, my_id, numprocs;
    long num_steps = 100000;
    double x, pi, mysteps, step, sum = 0.0;
    step = 1.0/(double) num_steps;
    MPI_Init(&argc, &argv);
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs);
    my_steps = num_steps/numprocs;  // divides num_steps among numprocs

    // each will get a bit of the range to do in its part of for loop
    for (i=my_id*my_steps; i<(my_id+1)*my_steps; i++)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

This is the MPI version Of the Monte-Carlo calculation on its own

---

Hybrid MPI + OpenMP Example 2 (/4):

**mixpi.c**

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

void main (int argc, char *argv[])
{
    int i, my_id, numprocs;
    long num_steps = 100000;
    double x, pi, mysteps, step, sum = 0.0;
    step = 1.0/(double) num_steps;
    MPI_Init(&argc, &argv);
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs);
    my_steps = num_steps/numprocs;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=my_id*my_steps; i<(my_id+1)*my_steps; i++)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

Get the MPI part done first, then add OpenMP pragma

---

Lecture 5: Message-Oriented Communication   CA4006 Lecture Notes (Martin Crane 2015)
Lecture Summary

- 2 Message Passing Primitives: Send & Receive with many different combos of each synch/asynch, persistent/transient
- Different types of procs in Message Passing: Peers/C+S/Filters
- Quite a lot of things to remember in MPI:
  - MPI programs need specific compilers (e.g. `mpicc`, `MPD`, `mpirun`).
  - Four functions for point-to-point comms, 6 more advanced ones, to synchronise, and perform collective comms,
- Unlike MPI, OpenMP:
  - Facilitates incremental parallelization of a serial program, so doesn’t require ‘all or nothing’ approach to parallelization,
  - MPI scales well but is non-trivial to implement for codes originally written for serial machines & not good for shared memory
  - Can implement both coarse-grain & fine-grain parallelism.
- Together, they can form a good team!