CS3006 Parallel and Distributed Computing

FALL 2022

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Chapter 7. Programming Shared Address Space Platforms

Synchronization Across Multiple for Directives

- Often, it is desirable to have a sequence of for-directives within a parallel construct that do not execute an implicit barrier at the end of each for directive.
- OpenMP provides a clause – `nowait`, which can be used with a for directive to indicate that the threads can proceed to the next statement without waiting for all other threads to complete the for loop execution.

Example 7.15 Using the nowait clause

Consider the following example in which variable `name` needs to be looked up in two lists – `current_list` and `past_list`. If the name exists in a list, it must be processed accordingly. The name might exist in both lists. In this case, there is no need to wait for all threads to complete execution of the first loop before proceeding to the second loop. Consequently, we can use the `nowait` clause to save idling and synchronization overheads as follows:

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i = 0; i < nmax; i++)
            if (isEqual(name, current_list[i])
                processCurrentName(name);
    #pragma omp for
        for (i = 0; i < mmax; i++)
            if (isEqual(name, past_list[i])
                processPastName(name);
}
```

The `sections` Directive

- OpenMP supports such non-iterative parallel task assignment using the `sections` directive.

- The general form of the sections directive is as follows:

```
#pragma omp sections [clause list]
{
   [#pragma omp section
      /* structured block */
   ]
   [#pragma omp section
      /* structured block */
   ]
  ...
}
```

The `sections` Directive

- This `sections` directive assigns the structured block corresponding to each section to one thread
  - (indeed more than one section can be assigned to a single thread).

- The clause list may include the following clauses – `private`, `firstprivate`, `lastprivate`, `reduction`, and `nowait`.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

Class Tasks

1. Write an OpenMP Program to display the difference between a STATIC Schedule and a Dynamic Schedule.

2. Write a Parallel C++ program to calculate WORDCOUNT of some text file. Your program must have reduction clause, and also display the local results of each thread

3. Repeat task#2 without using the REDUCTION clause.
   - **HINT: Single** directive

Nesting parallel Directives

```
#pragma omp parallel for default(private) shared (a, b, c, dim) \
                         num_threads(2)
    for (i = 0; i < dim; i++) {
    #pragma omp parallel for default(private) shared (a, b, c, dim) \
                         num_threads(2)
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            #pragma omp parallel for default(private) \
                         shared (a, b, c, dim) num_threads(2)
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

- The code as written only generates a logical team of threads on encountering a nested parallel directive.

- The newly generated logical team is still executed by the same thread corresponding to the outer `parallel` directive.

- To generate a new set of threads, nested parallelism must be enabled using the `OMP_NESTED` environment variable.
  - If the `OMP_NESTED` environment variable is set to `FALSE`, then the inner parallel region is serialized and executed by a single thread.
  - If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled.

Synchronization Constructs in OpenMP

- The need for coordinating the execution of multiple threads may be the result of a desired *execution order*, the *atomicity* of a set of instructions, or the need for *serial execution* of code segments.

- The Pthreads API supports **mutexes** and condition variables.

The `barrier` Directive

- OpenMP provides a `barrier` directive, whose syntax is as follows:

```
#pragma omp barrier
```

- On encountering this directive, all threads in a team wait until others have caught up, and then release.
- When used with nested parallel directives, the barrier directive binds to the closest parallel directive.
- For executing `barriers` conditionally, it is important to note that a barrier directive must be enclosed in a compound statement that is conditionally executed.

Single Thread Executions: The `single` and `master` Directives

- Often, a computation within a parallel section needs to be performed by just one thread.
  - A simple example of this is the computation of the **mean** of a list of numbers.
- A `single` directive specifies a structured block that is executed by a single (arbitrary) thread.
- The syntax of the single directive is as follows:

```
#pragma omp single [clause list]
    /* structured block */
```

Single Thread Executions: The `single` and `master` Directives

- On encountering the `single` block, the first thread enters the block. All the other threads proceed to the end of the block.

- If the `nowait` clause has been specified at the end of the block, then the other threads proceed; otherwise they wait at the end of the single block for the thread to finish executing the block.

Single Thread Executions: The `single` and `master` Directives

- The `master` directive is a specialization of the `single` directive in which only the master thread executes the structured block.

- The syntax of the master directive is as follows:

```
#pragma omp master
   /* structured block */
```

Critical Sections: The `critical` and `atomic` Directives

- In our discussion of Pthreads, we had examined the use of locks to protect critical regions
- OpenMP provides a `critical` directive for implementing critical regions.
- The syntax of a critical directive is:

```
#pragma omp critical [(name)]
    structured block
```

```
#pragma omp parallel sections
{
    #pragma parallel section
    {
        /* producer thread */
        task = produce_task();
        #pragma omp critical ( task_queue)
        {
            insert_into_queue(task);
        }
    }
    #pragma parallel section
    {
        /* consumer thread */
        #pragma omp critical ( task_queue)
        {
            task = extract_from_queue(task);
        }
        consume_task(task);
    }
}
```

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
  #pragma omp critical ( xaxis )
    x_next = dequeue(x);
  work(x_next);
  #pragma omp critical ( yaxis )
    y_next = dequeue(y);
  work(y_next);
}
```

Critical Sections: The `critical` and `atomic` Directives

- It is easy to see that the critical directive is a direct application of the corresponding **mutex** function in Pthreads.

- There are some obvious safeguards that must be noted while using the critical directive.
  - The block of instructions must represent a structured block, i.e., **no jumps** are permitted **into** or **out** of the block.
  - Jumping in would result in non-critical access and jumping out would result in an unreleased lock, which could cause the threads to wait indefinitely.

In-Order Execution: The `ordered` Directive

- In many circumstances, it is necessary to execute a *segment* of a parallel loop in the order in which the serial version would execute it.
  - For example, consider a for loop in which, at some point, we compute the cumulative sum in array `cumul_sum` of a list stored in array list.
  - The array cumul_sum can be computed using a for loop over index i serially by executing `cumul_sum[i] = cumul_sum[i-1] + list[i]`.
  - When executing this for loop across threads, it is important to note that `cumul_sum[i]` can be computed only after `cumul_sum[i-1]` has been computed.
  - Therefore, the statement would have to executed within an ordered block.

In-Order Execution: The `ordered` Directive

- The syntax of the ordered directive is as follows:

```
#pragma omp ordered
    structured block
```

- Since the ordered directive refers to the in-order execution of a `for` loop, it must be within the scope of a for or parallel `for` directive.

- Furthermore, the `for` or `parallel` for directive must have the ordered clause specified to indicate that the loop contains an `ordered` block.

- Ordered sections are useful for sequentially ordering the output from work that is done in parallel

```
#pragma omp for ordered schedule(dynamic)
  for (i=lb; i<ub; i+=st)
    work(i);

void work(int k)
{
  #pragma omp ordered
    printf(" %d", k);
}
```

Chapter 6. Programming Using the Message Passing Paradigm

Recall…

- A message-passing platform consists of ***p*** processing nodes, each with its own exclusive address space.
    - Interactions between processes running on different nodes must be accomplished using **messages** (data, work, and to synchronize actions among the processes), hence the name ***message passing***.
    - The basic operations in this programming paradigm are ***send*** and ***receive.***
    - The ***message-passing programming paradigm*** is one of the oldest and most widely used approaches for programming parallel computers.

Principles of Message-Passing Programming

- The two key attributes to characterize the message-passing programming paradigm.
  1. The first is that it assumes a **partitioned address space**
  2. Second is that it supports only **explicit parallelization**.

- There are two immediate implications of a partitioned address space.
  1. First, each data element must belong to one of the partitions of the space; hence, **data must be explicitly partitioned and placed**.
  2. The second implication is that all interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data

Principles of Message-Passing Programming

- Message-passing programs are often written using the ***asynchronous*** or ***loosely synchronous*** paradigms
  - In the *asynchronous paradigm*, all concurrent tasks execute asynchronously.
  - However, such programs can be harder to reason about, and can have nondeterministic behavior due to race conditions .
  - In the *loosely synchronous model*, tasks or subsets of tasks synchronize to perform **interactions**. Between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the ***single program multiple data*** (SPMD) model.

The Building Blocks: **Send** and **Receive** Operations

- In their simplest form, the prototypes of these operations are defined as follows:

```
send(void *sendbuf, int nelems, int dest)

receive(void *recvbuf, int nelems, int source)
```

```
P0                                  P1

a = 100;                            receive(&a, 1, 0)
send(&a, 1, 1);                     printf("%d\n", a);
a=0;
```

- The semantics of the send operation require that the value received by process **P1** must be **100** as opposed to **0**.

- This motivates the design of the **send** and **receive protocols**.

MPI: the Message Passing Interface (Chapter#6)

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.

- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.

- The MPI library contains over 125 routines, but the number of key concepts is much smaller, it is possible to write fully-functional message-passing programs by using only the six routines
  - These routines are used to *initialize* and *terminate* the MPI library, to get information about the parallel computing environment, and to *send* and *receive* messages.

MPI: the Message Passing Interface

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Starting and Terminating MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines.
  - Its purpose is to initialize the MPI environment.

- Calling `MPI_Init` more than once during the execution of a program will lead to an error.

- `MPI_Finalize` is called at the end of the computation, and it performs various cleanup tasks to terminate the MPI environment.

- No MPI calls may be performed after `MPI_Finalize` has been called, not even `MPI_Init`

- The prototypes of these two functions are:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- All MPI routines, data-types, and constants are prefixed by “`MPI_`”.

Communicators

- A key concept used throughout MPI is that of the *communication domain*.
- A **communication domain** is a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`, that are called ***communicators*** .

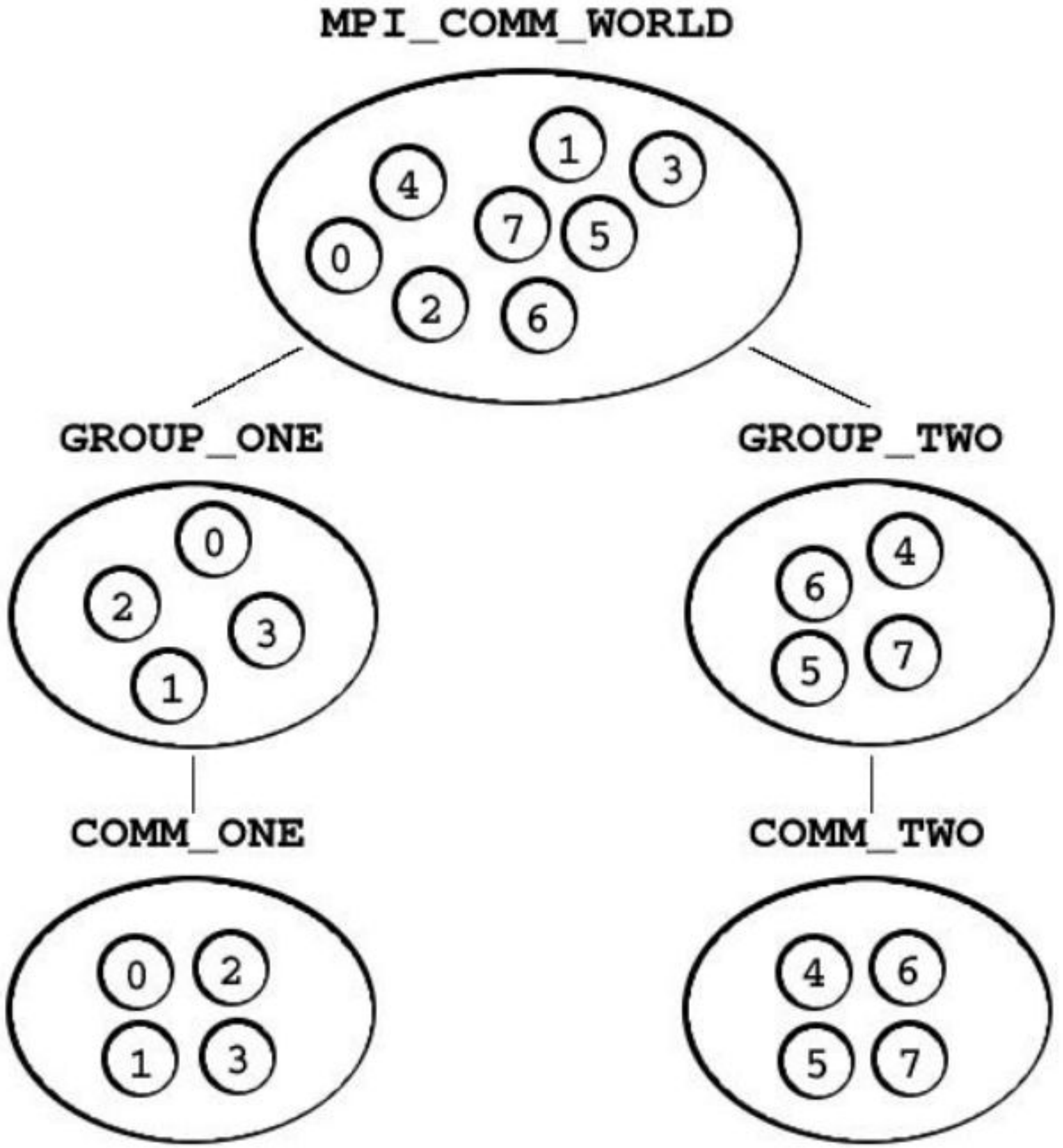
Communicators

- A process can belong to many different (possibly overlapping) communication domains

- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Communicators

- Communicator holds a group of processes that can communicate with each other.
- All message passing calls in MPI must have a specific communicator to use the call with.
- An example of a communicator handle is MPI_COMM_WORLD. MPI_COMM_WORLD is the default communicator that contains all processes available for use.
- Can be created and destroyed during program execution.



MPI_COMM_WORLD
GROUP_ONE
GROUP_TWO
COMM_ONE
COMM_TWO

Getting Information

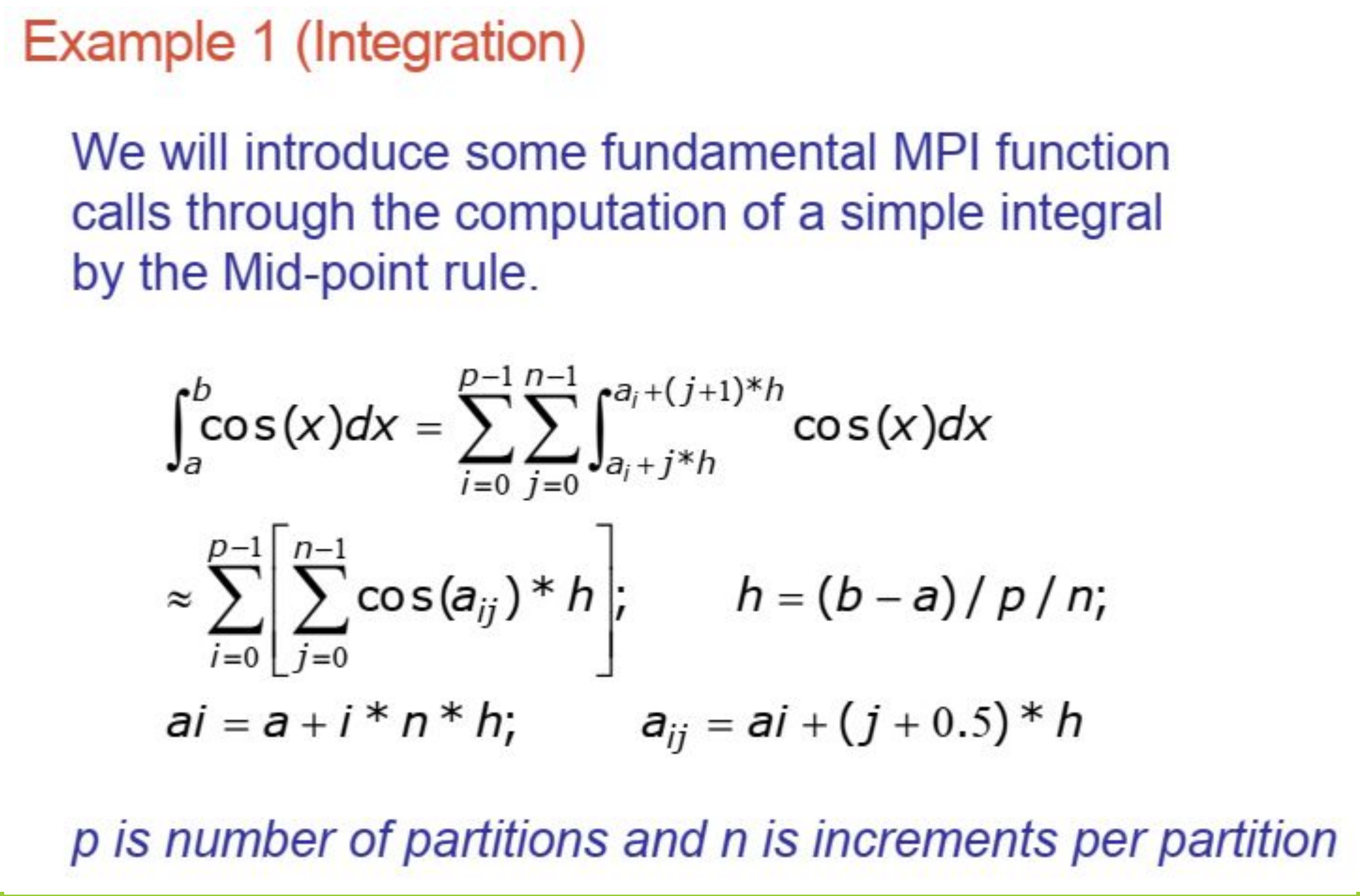
- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.

- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Lab



Example 1 (Integration)

We will introduce some fundamental MPI function calls through the computation of a simple integral by the Mid-point rule.

$$\int_a^b \cos(x)dx = \sum_{i=0}^{p-1}\sum_{j=0}^{n-1}\int_{a_i+j*h}^{a_i+(j+1)*h} \cos(x)dx$$

$$\approx \sum_{i=0}^{p-1}\left[\sum_{j=0}^{n-1}\cos(a_{ij})*h\right]; \qquad h = (b-a)/p/n;$$

$$ai = a + i*n*h; \qquad a_{ij} = ai + (j+0.5)*h$$

p is number of partitions and n is increments per partition

Example 1 - Serial C code

```
#include <math.h>
#include <stdio.h>
float integral(float a, int i, float h, int n);
void main() {
    int n, p, i, j, ierr;
    float h, integral_sum, a, b, pi, ai;
    pi = acos(-1.0);  /* = 3.14159... *
    a = 0.;           /* lower limit of integration */
    b = pi/2.;        /* upper limit of integration */
    p = 4;            /* # of partitions */
    n = 500;          /* increments in each process */
    h = (b-a)/n/p;    /* length of increment */
    integral_sum = 0.0;
    for (i=0; i<p; i++) { /* integral sum over partitions */
      ai = a + i*n*h;    /* lower limit of int. for partition i */
      integral_sum += integral(ai,h,n); }
    printf("The Integral =%f\n", integral_sum);
}
```

```
float integral(float ai, float h, int n) {
  int j;
  float aij, integ;
  integ = 0.0;                    /* initialize integral */
  for (j=0; j<n; j++) {           /* sum over integrals in partition i*/
    aij = ai + (j+0.5)*h;         /* lower limit of integration of j*/
    integ += cos(aij)*h;          /* contribution due j */
  }
  return integ;
}
```

Example 1_1 - Parallel C code

Two main styles of programming: SPMD, MPMD. The following demonstrates SPMD, which is more frequently used than MPMD,

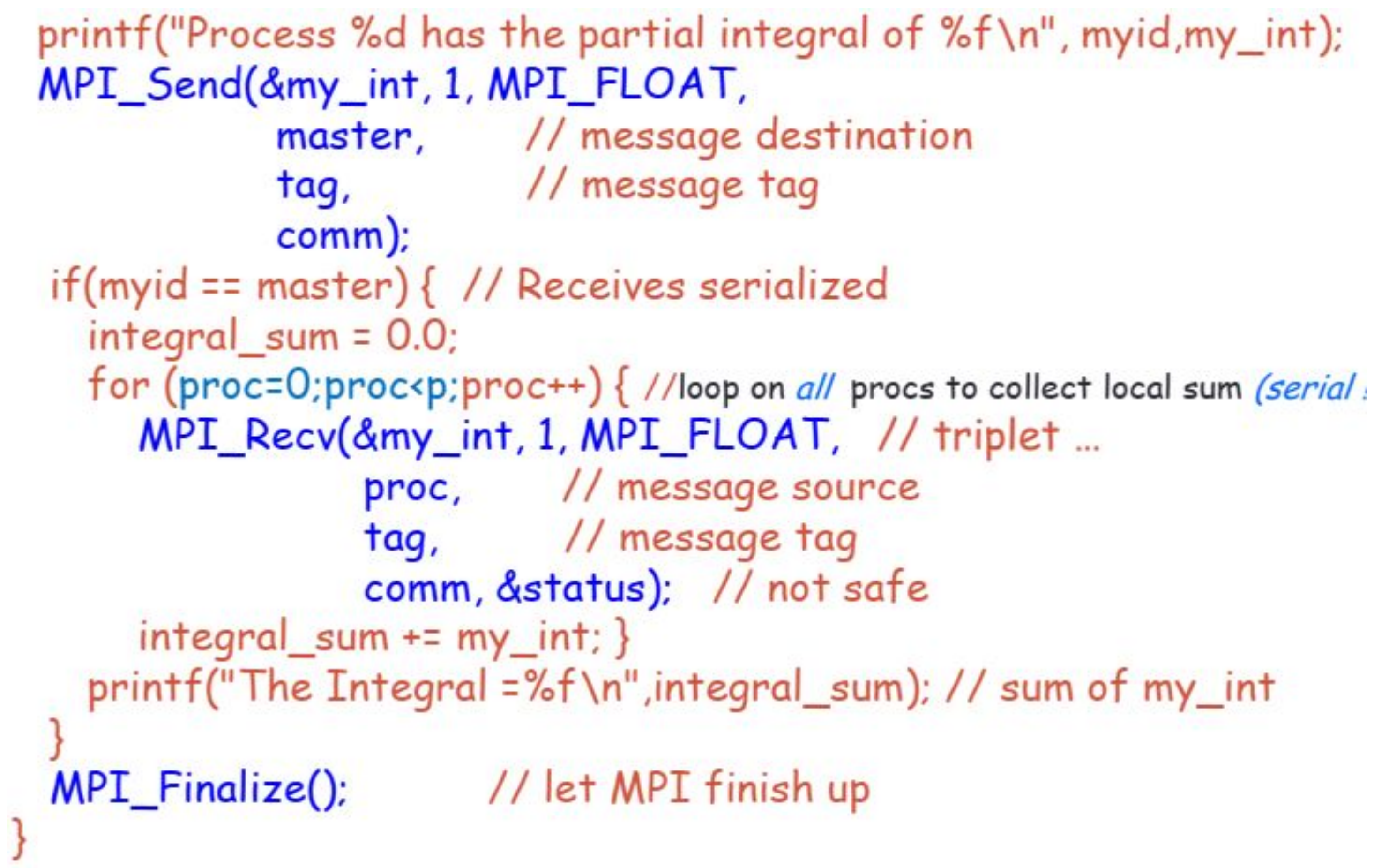
MPI functions used in this example:

- MPI_Init, MPI_Comm_rank, MPI_Comm_size
- MPI_Send, MPI_Recv, MPI_Finalize

```
#include <mpi.h>
float integral(float ai, float h, int n);   // prototyping
void main(int argc, char* argv[])
{
   int n, p, myid, tag, proc, ierr;
   float h, integral_sum, a, b, ai, pi, my_int;
   int master = 0;  /* processor performing total sum */
   MPI_Comm comm;
   MPI_Status status;
```

```
comm = MPI_COMM_WORLD;
ierr = MPI_Init(&argc,&argv);          // starts MPI
MPI_Comm_rank(comm, &myid);            // get current process id
MPI_Comm_size(comm, &p);               // get number of processes

pi = acos(-1.0);       // = 3.14159...
a = 0.;                // lower limit of integration
b = pi*1./2.;          // upper limit of integration
n = 500;               // number of increment within each process
tag = 123;             // set the tag to identify this particular job
h = (b-a)/n/p;         // length of increment
ai = a + myid*n*h;  // lower limit of integration for partition myid
my_int = integral(ai, h, n)    // compute local sum due myid
```



```
  printf("Process %d has the partial integral of %f\n", myid,my_int);
  MPI_Send(&my_int, 1, MPI_FLOAT,
               master,        // message destination
               tag,           // message tag
               comm);
   if(myid == master) {  // Receives serialized
     integral_sum = 0.0;
     for (proc=0;proc<p;proc++) { //loop on all procs to collect local sum (serial
        MPI_Recv(&my_int, 1, MPI_FLOAT,   // triplet ...
                    proc,       // message source
                    tag,        // message tag
                    comm, &status);   // not safe
        integral_sum += my_int; }
     printf("The Integral =%f\n",integral_sum); // sum of my_int
   }
   MPI_Finalize();          // let MPI finish up
}
```

Our First MPI Program

```
#include <mpi.h>

main(int argc, char *argv[])
{
   int npes, myrank;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   printf("From process %d out of %d, Hello World!\n",myrank, npes);
   MPI_Finalize();
}
```

Lab Task(s)

1. Write a Parallel C++ program with OpneMP to calculate WORDCOUNT of some text file. Your program must have reduction clause, and also display the local results of each thread.

2. Download, Install and Configure MPI. Write a MPI C program to print the statement "Hello, I am process X of Y processes“
   - where X is the current process while Y is the number of processes for job

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Running an MPI Program

```
mpicc program_name.c -o object_file


mpirun -np [number of processes] ./object_file
```

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv` respectively.

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

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

MPI_Send

- MPI_Send sends the data stored in the buffer pointed by
  - buf :initial address of send buffer (choice).
  - count:number of elements in send buffer (nonnegative integer).
  - datatype: datatype of each send buffer element (handle)
  - dest: rank of destination (integer)
  - tag: message tag (integer), each message has an integer-valued tag associated with it to distinguish different types of messages.
  - comm: communicator (handle)

- The dest argument is the rank of the destination process in the communication domain specified by the communicator comm .

MPI_Recv

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

- `MPI_Recv` receives a message sent by a process whose rank is given by the `source` in the communication domain specified by the `comm` argument.
- The received message is stored in continuous locations in the buffer pointed to by `buf` .
- The tag of the sent message must be that specified by the `tag` argument.
  - If there are many messages with identical tag from the same process, then any one of these messages is received

MPI_Recv

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

- MPI allows specification of *wildcard* arguments for both `source` and `tag`.
  - If `source` is set to `MPI_ANY_SOURCE` , then any process of the communication domain can be the source of the message.
  - Similarly, if `tag` is set to `MPI_ANY_TAG` , then messages with any tag are accepted.

MPI_Recv

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

- The `count` and `datatype` arguments of `MPI_Recv` are used to specify the length of the supplied buffer.
  - The received message should be of length equal to or less than this length.

`MPI_Recv`

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

- After a message has been received, the `status` variable can be used to get information about the `MPI_Recv` operation.

- In C, `status` is stored using the `MPI_Status` data-structure.

- The corresponding data structure contains:

```
typedef struct MPI_Status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR; };
```

- MPI_SOURCE and MPI_TAG store the source and the tag of the received message.
- They are particularly useful when MPI_ANY_SOURCE and MPI_ANY_TAG are used for the source and tag arguments.
- MPI_ERROR stores the error-code of the received message.