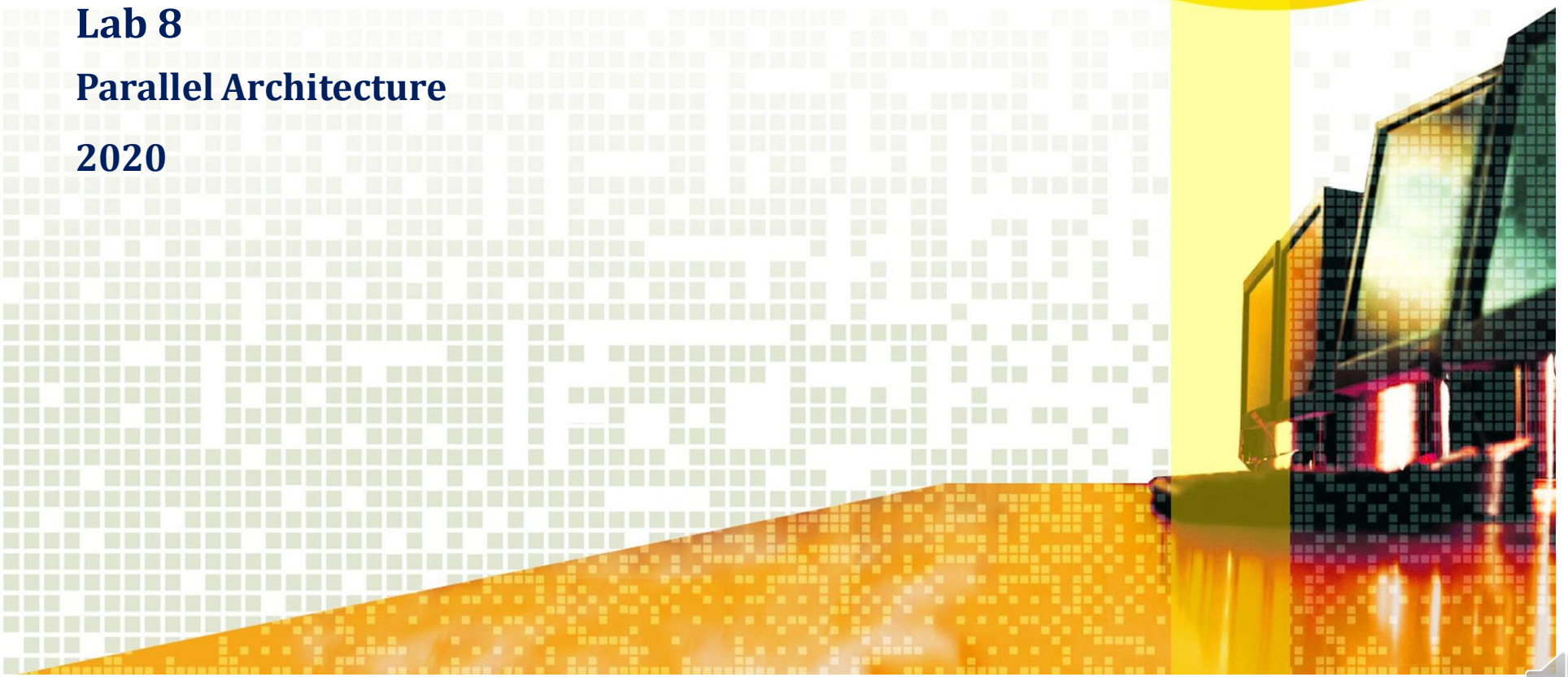
OpenMP/MPI

Lab 8

Parallel Architecture

2020



Open MP Clauses/Directive

Clauses / Directives Summary

- The table below summarizes which clauses are accepted by which OpenMP directives.

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Open MP Clauses/Directive

- The following OpenMP directives do not accept clauses:
  - MASTER
  - CRITICAL
  - BARRIER
  - ATOMIC
  - FLUSH
  - ORDERED
  - THREADPRIVATE

Directive Binding and Nesting Rules

Directive Binding:

- The DO/for, SECTIONS, SINGLE, MASTER and BARRIER directives bind to the dynamically enclosing PARALLEL, if one exists. I directives have no effect.
- The ORDERED directive binds to the dynamically enclosing DO/for.
- The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.
- The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL.

Run-time Library routines

- The OpenMP API includes an ever-growing number of run-time library routines.
- These routines are used for a variety of purposes as shown in the table below:

Routine	Purpose
OMP_SET_NUM_THREADS	Sets the number of threads that will be used in the next parallel region
OMP_GET_NUM_THREADS	Returns the number of threads that are currently in the team executing the parallel region from which it is called
OMP_GET_MAX_THREADS	Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function
OMP_GET_THREAD_NUM	Returns the thread number of the thread, within the team, making this call.
OMP_GET_THREAD_LIMIT	Returns the maximum number of OpenMP threads available to a program
OMP_GET_NUM_PROCS	Returns the number of processors that are available to the program
OMP_IN_PARALLEL	Used to determine if the section of code which is executing is parallel or not
OMP_SET_DYNAMIC	Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions
OMP_GET_DYNAMIC	Used to determine if dynamic thread adjustment is enabled or not

Run-time Library routines

OMP_IN_PARALLEL	Used to determine if the section of code which is executing is parallel or not
OMP_SET_DYNAMIC	Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions
OMP_GET_DYNAMIC	Used to determine if dynamic thread adjustment is enabled or not
OMP_SET_NESTED	Used to enable or disable nested parallelism
OMP_GET_NESTED	Used to determine if nested parallelism is enabled or not
OMP_SET_SCHEDULE	Sets the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_GET_SCHEDULE	Returns the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_SET_MAX_ACTIVE_LEVELS	Sets the maximum number of nested parallel regions
OMP_GET_MAX_ACTIVE_LEVELS	Returns the maximum number of nested parallel regions
OMP_GET_LEVEL	Returns the current level of nested parallel regions
OMP_GET_ANCESTOR_THREAD_NUM	Returns, for a given nested level of the current thread, the thread number of ancestor thread
OMP_GET_TEAM_SIZE	Returns, for a given nested level of the current thread, the size of the thread team
OMP_GET_ACTIVE_LEVEL	Returns the number of nested, active parallel regions enclosing the task that contains the call
OMP_IN_FINAL	Returns true if the routine is executed in the final task region; otherwise it returns false
OMP_INIT_LOCK	Initializes a lock associated with the lock variable
OMP_DESTROY_LOCK	Disassociates the given lock variable from any locks
OMP_SET_LOCK	Acquires ownership of a lock
OMP_UNSET_LOCK	Releases a lock
OMP_TEST_LOCK	Attempts to set a lock, but does not block if the lock is unavailable
OMP_INIT_NEST_LOCK	Initializes a nested lock associated with the lock variable

Environment Variables

- OpenMP provides the following environment variables for controlling the execution of parallel code.
- All environment variable names are uppercase. The values assigned to them are not case sensitive.

OMP_SCHEDULE

Applies only to DO, PARALLEL DO (Fortran) and `for, parallel for` (C/C++) directives which have their schedule clause set to RUNTIME. how iterations of the loop are scheduled on processors. For example:

```
setenv OMP_SCHEDULE "guided, 4"
setenv OMP_SCHEDULE "dynamic"
```

OMP_NUM_THREADS

Sets the maximum number of threads to use during execution. For example:

```
setenv OMP_NUM_THREADS 8
```

OMP_DYNAMIC

Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE.

```
setenv OMP_DYNAMIC TRUE
```

Environment Variables

OMP_PROC_BIND

Enables or disables threads binding to processors. Valid values are TRUE or FALSE. For example:

```
setenv OMP_PROC_BIND TRUE
```

OMP_NESTED

Enables or disables nested parallelism. Valid values are TRUE or FALSE. For example:

```
setenv OMP_NESTED TRUE
```

OMP_STACKSIZE

Controls the size of the stack for created (non-Master) threads. Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

Message-Passing Programming

- Abstraction of Multi-Computers system.

- MPI (Message Passing Interface) is the standard programming interface, that defines a set of functions that allow a programmer to instruct their code to execute tasks in parallel.

Basics

Initialization

- All MPI programs must begin with a call to

  MPI::Init( ) //Initialize the MPI execution environment

- And close with a call to

  MPI::Finalize( ) // Terminates MPI execution environment

Basics

Size and rank

- Get how many processes are running in a given *communicator*,

```
size= MPI::COMM_WORLD.Get_size();
```

- and the rank of the calling process within that communicator.

```
myrank= MPI::COMM_WORLD.Get_rank();
```

Point-to-Point Communication

```
MPI::COMM_WORLD.Send(   void *buf,
                        int count,
                        MPI_Datatype datatype,
                        int dest,
                        int tag)
```

MPI_Send(&x, 1, MPI_DOUBLE, dest, mytag)

```
MPI::COMM_WORLD.Recv (  void *buf,
                        int count,
                        MPI_Datatype datatype,
                        int source,
                        int tag)
```

MPI_Receive(&x, 1, MPI_DOUBLE, source, mytag)

Send & Receive process

Blocking	Non-Blocking
Send | Recv	**Isend | Irecv**
Blocking point-to-point operations will wait until a communication has completed on its local processor before continuing. (Subroutine will not return till copy is completed to/from system buffer)	Non-blocking point-to-point operations will initiate a communication without waiting for that communication to be completed. (Copy is just initiated to/from system buffer). You must handle waiting for correct data by yourself.

Send Array

```
int MyID , Size , Sender , Receiver ;
MPI::Init( Argc , Argv);
MyID = MPI::COMM_WORLD.Get_rank();
Size = MPI::COMM_WORLD.Get_size();
if(MyID == 0)
{
    int Data[10] = {1,2,3,4,5,6,7,8,9,10};
    Receiver = 1;
    MPI::COMM_WORLD.Send( &Data , 10 , MPI::INT ,Receiver, 0);
}
else
{
    int Data[10];
    Sender = 0;
    MPI::COMM_WORLD.Recv( &Data , 10 , MPI::INT ,Sender, 0);
    for(int i =0;i<10;i++)
        cout<<Data[i]<<endl;
}
MPI::Finalize();
return 0;
```

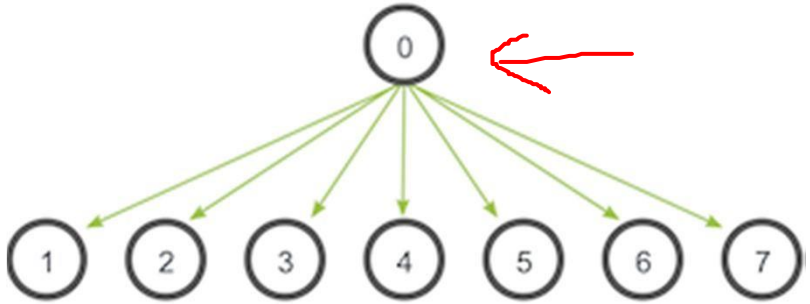
Collective Communication

- Collective communication involves all processes in the scope of the communicator.
- All processes are by default, members in the communicator MPI_COMM_WORLD.

Collective Communication - Broadcast

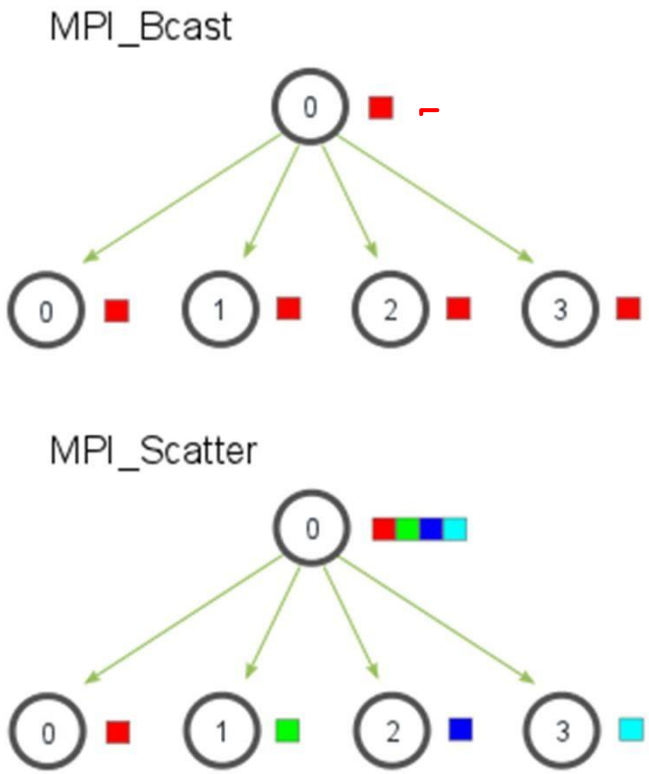
```
MPI::Comm::Bcast (      void* buffer,
                        int count,
                        const MPI::Datatype& datatype,
                        int root
                  )
```

- INOUT buffer starting address of buffer (choice)
- IN count number of entries in buffer (non-negative integer)
- IN datatype data type of buffer (handle)
- IN root rank of broadcast root (integer)



Collective Communication - Scatter

- MPI_Scatter is a collective routine that is very similar to MPI_Bcast
- MPI_Scatter involves a designated root process sending data to all processes in a communicator.

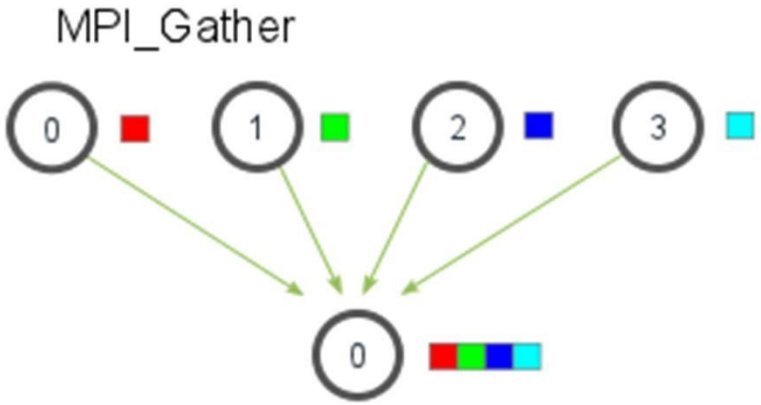


Collective Communication - Scatter

- **MPI::Comm::Scatter(** **void* send_data,**
  int send_count,
  MPI_Datatype send_datatype,
  void* recv_data,
  int recv_count,
  MPI_Datatype recv_datatype,
  int root)

Collective Communication - Gather

- MPI_Gather is the inverse of MPI_Scatter. Instead of spreading elements from one process to many processes, MPI_Gather takes elements from many processes and gathers them to one single process.



Collective Communication - Gather

- **MPI::Comm::**Gather(void* send_data,
  int send_count,
  MPI_Datatype send_datatype,
  void* recv_data,
  int recv_count,
  MPI_Datatype recv_datatype,
  int root)

Collective Communication - Reduction

- **Reduction** Perform an operation over data on all processes and store the result in one process

- **MPI::Comm::Reduce(**

```
void* sendbuf,
void* recvbuf,
int count,
const MPI::Datatype& datatype,
const MPI::Op& op,
int root)
```

- IN-- **sendbuf** address of send buffer (choice)
- OUT-- **recvbuf** address of receive buffer (choice, significant only at root)
- IN-- **count** number of elements in send buffer (non-negative integer)
- IN-- **datatype** data type of elements of send buffer (handle)
- IN-- **op** reduce operation (handle)
- IN-- **root** rank of root process (integer)

Collective Communication - Reduction

MPI::Comm::Reduce operators

MPI::MAX
MPI::MIN
MPI::SUM
MPI::PROD
MPI::MAXLOC
MPI::MINLOC
MPI::LAND
MPI_BAND
MPI::LOR
MPI::BOR
MPI::LXOR
MPI::BXOR

MPI_Wtime

Returns an elapsed time on the calling processor

double MPI_Wtime(void)

Return value : Time in seconds since an arbitrary time in the past.

Speedup

Speedup: $S_p = \frac{T_s}{T_p}$

- p = # of processors
- Ts = execution time of the sequential algorithm
- Tp = execution time of the parallel algorithm with p processors
- $Sp = P$ (linear speedup: ideal)

Parallel efficiency

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$$

What is Hybridization?

- the use of inherently different models of programming in a complementary manner, in order to achieve some benefit not possible otherwise;
- a way to use different models of parallelization in a way that takes advantage of the good points of each;
- also known as "mixed mode" programming

When Does Hybridization Make Sense?

- when one wants to scale a shared memory OpenMP application for use on multiple SMP nodes in a cluster;
- when one wants to reduce an MPI application's sensitivity to becoming communication bound;
- when one is designing a parallel program from the very beginning to maximize utilization of a distributed memory machine consisting of individual SMP nodes;

Hybridization Using MPI and OpenMP

- facilitates cooperative shared memory (OpenMP) programming across clustered SMP nodes;
- MPI facilitates communication among SMP nodes, including the efficient packing and sending of complex data structures;
- OpenMP manages the workload on each SMP node;
- MPI and OpenMP are used in tandem to manage the overall concurrency of the application;

MPI vs. OpenMP

- **Pure MPI Pro:**
  - High scalability
  - High portability
  - No false sharing
  - Scalability out-of-node
- **Pure MPI Con:**
  - Hard to develop and debug.
  - Explicit communications
  - Coarse granularity
  - Hard to ensure load balancing

Pure OpenMP Pro:

- Easy to deploy (often)
- Low latency
- Implicit communications
- Coarse and fine granularity
- Dynamic Load balancing

Pure OpenMP Con:

- Only on shared memory machines
- Intranode scalability
- Possible long waits for unlocking data
- Undefined thread ordering

Example: Calculating π

- **Numerical integration**

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

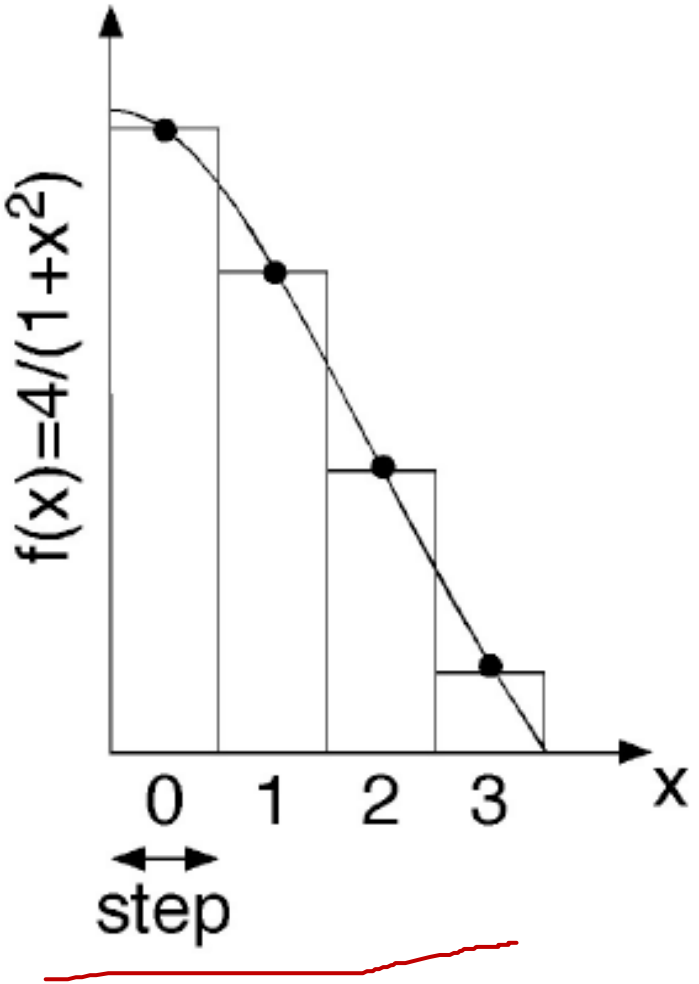
- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`

$x_i = (i+0.5)\Delta$ $(i = 0,\ldots,N\text{-}1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\Delta \cong \pi$$

```
#include <stdio.h>
#define NBIN 100000
void main() {
  int i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);}
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```



pi – MPI version

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>              /* MPI header file */
#define NUM_STEPS 100000000

int main(int argc, char *argv[]) {
    int nprocs;
    int myid;
    double start_time, end_time;
    int i;
    double x, pi;
    double sum = 0.0;
    double step = 1.0/(double) NUM_STEPS;

    /* initialize for MPI */
   MPI_Init(&argc, &argv);       /* starts MPI */
    /* get number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    /* get this process's number (ranges from 0 to nprocs - 1) */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
    /* do computation */
    for (i=myid; i < NUM_STEPS; i += nprocs) {      /* changed */
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    sum = step * sum;                               /* changed */
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);/* added */

/* print results */
    if (myid == 0) {
        printf("parallel program results with %d processes:\n", nprocs);
        printf("pi = %g  (%17.15f)\n",pi, pi);
    }

    /* clean up for MPI */
    MPI_Finalize();

    return 0;
}
```

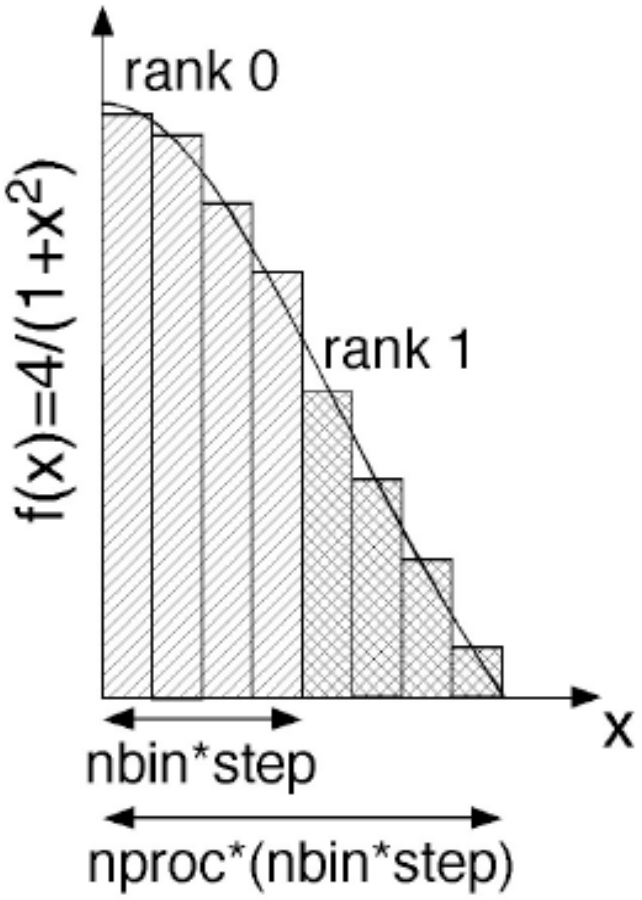
OpenMP, reduction clause

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
int main(int argc, char *argv[ ]) {
    int I, nthreads;
    double x, pi;
    double sum = 0.0;
    double step = 1.0/(double) NUM_STEPS;

    /* do computation -- using all available threads */
    #pragma omp parallel
    {
        #pragma omp for private(x) reduction(+:sum) schedule(runtime)
        for (i=0; i < NUM_STEPS; ++i) {
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
        #pragma omp master
        {
            pi = step * sum;
        }
    }
    printf("PI = %f\n",pi);
}
```

MPI+OpenMP Calculation of π

- Each MPI process integrates over a range of width `1/nproc`, as a discrete sum of `nbin` bins each of width `step`
- Within each MPI process, `nthreads` OpenMP threads perform part of the sum as in `omp_pi.c`



MPI_OpenMP version

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>              /* MPI header file */
#include <omp.h>              /* OpenMP header file */
#define NUM_STEPS 100000000
#define MAX_THREADS 4

int main(int argc, char *argv[]) {
    int nprocs, myid;
    int tid, nthreads, nbin;
    double start_time, end_time;
    double pi, Psum=0.0, sum[MAX_THREADS]={0.0};
    double step = 1.0/(double) NUM_STEPS;

    /* initialize for MPI */
   MPI_Init(&argc, &argv);       /* starts MPI */
    /* get number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    /* get this process's number (ranges from 0 to nprocs - 1) */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    nbin= NUM_STEPS/nprocs;
```

```
#pragma omp parallel private(tid)
{
        int i;
        double x;
        nthreads=omp_get_num_threads();
        tid=omp_get_thread_num();
        for (i=nbin*myid+tid; i < nbin*(myid+1); i+= nthreads) { /* changed*/
            x = (i+0.5)*step;
            sum[tid] += 4.0/(1.0+x*x);
        }
}
 for(tid=0; tid<nthreads; tid++)   /*sum by each mpi process*/
        Psum += sum[tid]*step;

MPI_Reduce(&Psum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);/* added */

    if (myid == 0) {
        printf("parallel program results with %d processes:\n", nprocs);
        printf("pi = %g  (%17.15f)\n",pi, pi);
    }
    MPI_Finalize();

    return 0;
}
```

Assignment

30x30 matrix multiplication
-OpenMP only
-MPI only
Hyprid OpenMp+MPI
-Analysis Study (in time)

Thank You