Introduction to OpenMP

Le Yan
HPC User Services @ LSU

LSU CENTER FOR COMPUTATION & TECHNOLOGY

LSU INFORMATION TECHNOLOGY SERVICES

Parallel Computing

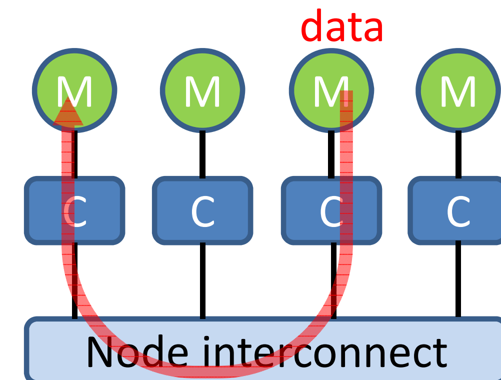
- Multiple processing units (workers) work together to process a workload
  - The processing units can be tightly or loosely coupled
  - In most cases, communication (data sharing) among workers is necessary for the purpose of coordination

Sharing Data among Workers

- There are a few different memory models
  - Distributed Memory
  - Shared Memory
  - Other memory models
    - Hybrid model
    - PGAS (Partitioned Global Address Space)

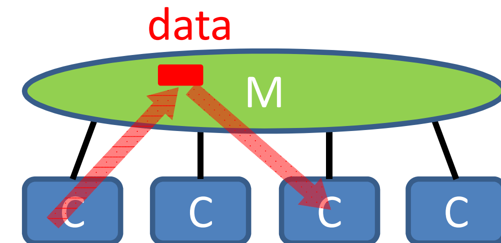
Distributed Memory Model

- Each process has its own address space
  - Data is local to each process
- Data sharing achieved via explicit message passing (through network)
- Example: MPI (Message Passing Interface)



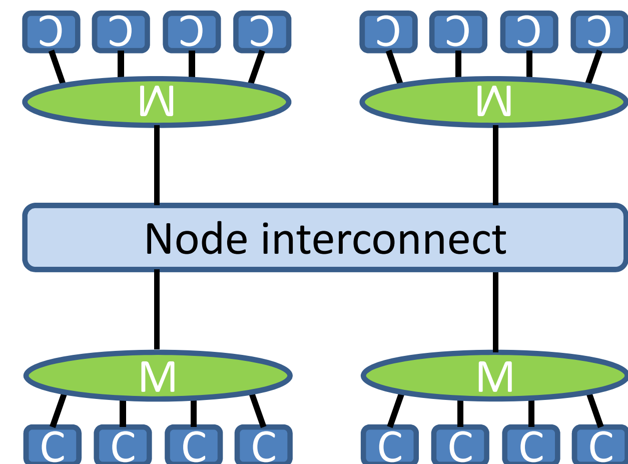
Shared Memory Model

- All threads can access the global address space
- Data sharing achieved via writing to/reading from the same memory location
- Example: OpenMP



Clusters of SMP Nodes

- Clusters of SMP (symmetric multi-processing) nodes dominate nowadays
- Hybrid model matches the physical structure of SMP clusters
  - OpenMP within nodes
  - MPI between nodes



LSU

LSU INFORMATION TECHNOLOGY SERVICES

Distributed vs. Shared Memory

Distributed

- Pro
  - Memory amount is scalable
  - Cheaper to build
- Con
  - Slow data sharing
    - Hard to balance the load
- ?
  - Explicit data transfer

Shared

- Pro
  - Easy to use
  - Fast data sharing
- Con
  - Memory amount is not scalable
  - Expensive to build
- ?
  - Implicit data transfer

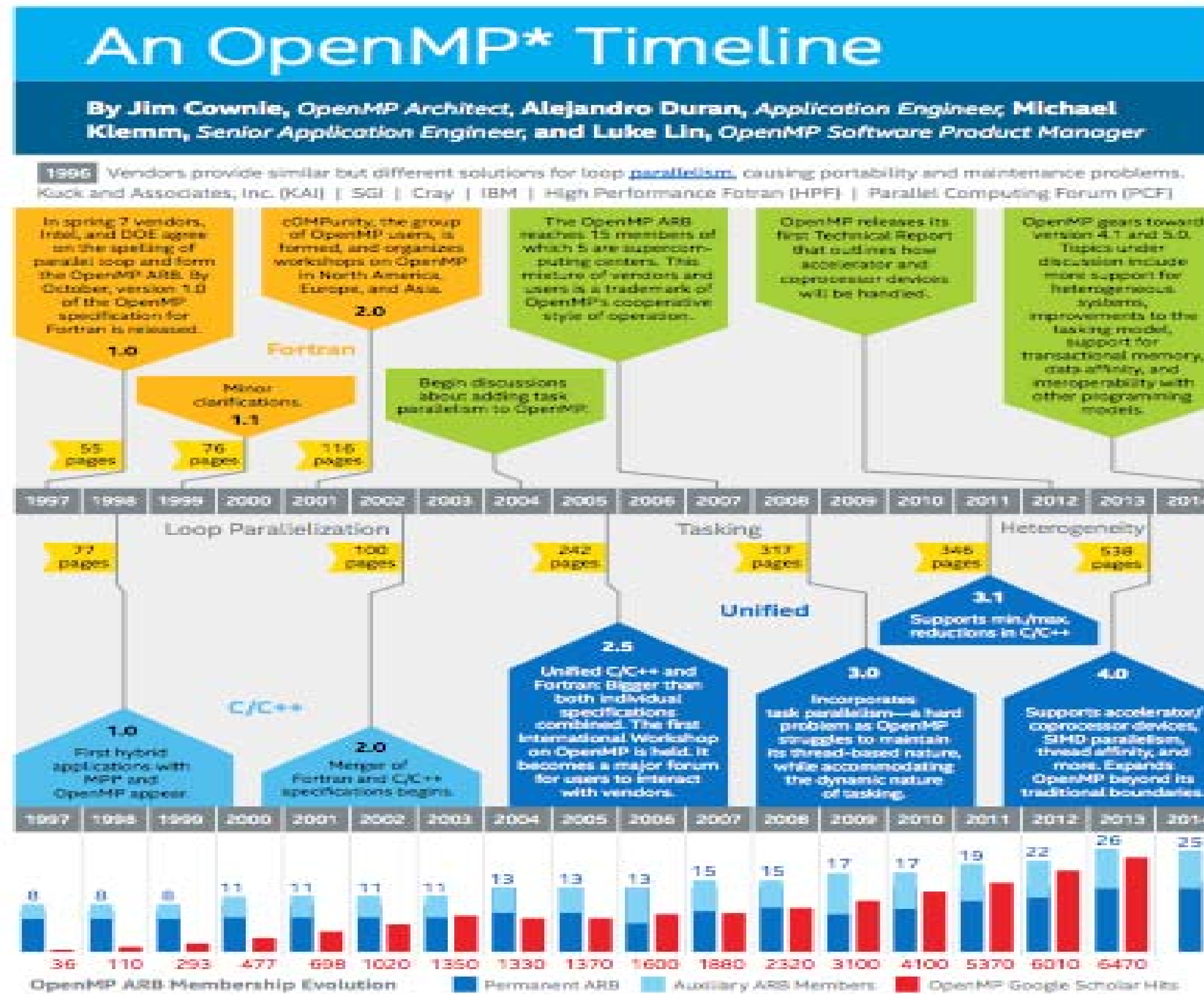
LSU CENTER FOR COMPUTATION & TECHNOLOGY

LONI

OpenMP

- OpenMP is an Application Program Interface (API) for thread based parallelism
- Supports Fortran, C and C++
- Uses a fork-join execution model
- OpenMP structures are built with program directives, runtime libraries and environment variables
- OpenMP has been the industry standard for shared memory programming over the last decade
- OpenMP 4.0 released in 2013
  - Work on 5.0 already started

OpenMP Timeline



Why OpenMP

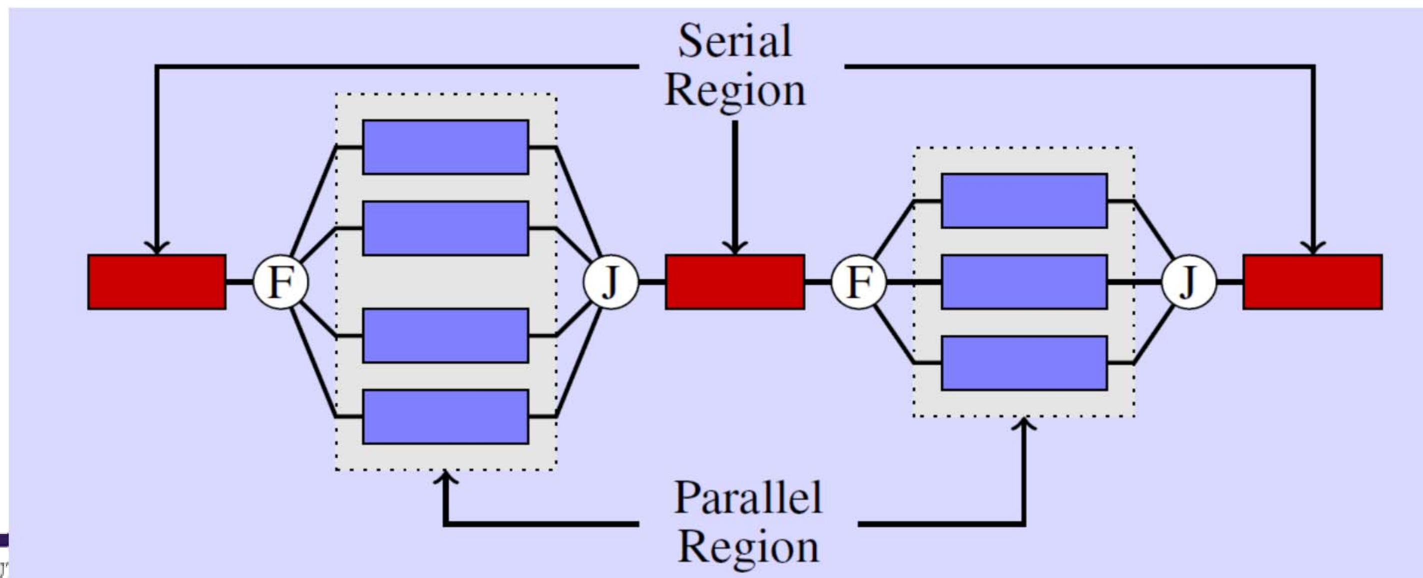
- Portability
  - Standard among many shared memory platforms
  - Implemented in major compiler suites
- Ease to use
  - Serial programs can be parallelized by adding compiler Directives
  - Allows for incremental parallelization – a serial program evolves into a parallel program by parallelizing different sections incrementally

LSU

LSU INFORMATION TECHNOLOGY SERVICES

Fork and Join Model

- Parallelism is achieved by generating multiple threads that run in parallel
  - A fork (F) is when a single thread is made into multiple, concurrently executing threads
  - A join (J) is when the concurrently executing threads synchronize back into a single thread
- OpenMP programs essentially consist of a series of forks and joins.



LSU CENTER FOR COMPUTATION & TECHNOLOGY

Building Blocks of OpenMP

- Program directives
  - Parallel regions
  - Parallel loops
  - Synchronization
  - Data structure
  - ...
- Runtime library routines
- Environment variables

OpenMP Basic Syntax

- Fortran (case insensitive)
  - Add: `use omp_lib` or `include "omp_lib.h"`
  - Usage: `sentinel directive [clauses]`
  - Fortran 77
    - Sentinel could be: `!$omp`, `*$omp`, `c$omp` and must begin in first column
  - Fortran 90/95/2003
    - Sentinel: `!$omp`
  - End of parallel region is signified by the end sentinel statement: `!$omp end directive [clauses]`
- C/C++ (case sensitive)
  - Add: `#include <omp.h>`
  - Usage: `#pragma omp directive [clauses]`

The First Directive - Parallel

- The `parallel` directive forms a group of threads for parallel execution.
- Each thread executes the block of code within the OpenMP parallel region.

Hello World Fortran

```
program hello

use omp_lib
implicit none
integer :: omp_get_thread_num, omp_get_num_threads

!$omp parallel
print *, 'Hello from thread',omp_get_thread_num() , &
'out of ' omp_get_num_threads(),' threads'
!$omp end parallel

end program hello
```

Start and end of the parallel region

Runtime functions

```
Hello from thread 0 out of 4 threads
Hello from thread 1 out of 4 threads
Hello from thread 2 out of 4 threads
Hello from thread 3 out of 4 threads
```

Hello World C

```
#include <omp.h>
#include <stdio.h>
int main () {

#pragma omp parallel
{
printf("Hello from thread %d out of %d threads\n”,omp_get_thread_num() ,
omp_get_num_threads() );
}

return 0;

}
```

```
Hello from thread 0 out of 4 threads
Hello from thread 1 out of 4 threads
Hello from thread 2 out of 4 threads
Hello from thread 3 out of 4 threads
```

Compilation and Execution

- Compile
  - Syntax: <compiler> <OpenMp flag> <source file>
  - On LSU and LONI HPC machines
    - Intel: `(ifort|icc)` **`-openmp`** `hello.(f90|c)`
    - PGI: `(pgfortran|pgcc)` **`-mp`** `hello.(f90|c)`
    - GNU: `(gfortran|gcc)` **`-fopenmp`** `hello.(f90|c)`
- Execute
  - `[OMP_NUM_THREADS=<# of threads>] <executable>`

Exercise 1: Hello World

- Write a “hello world” program with OpenMP where
  - If the thread id is odd, then print a message "Hello world from thread x, I’m odd!”
  - If the thread id is even, then print a message "Hello world from thread x, I’meven!”

Exercise 1: Hello World

C

```c
#include <stdio.h>
/* Include omp.h ? */
int main() {
/* Add Opemp pragma */
  {
  if (id%2==1)
    printf("Hello world from thread %d,
    I am odd\n", /* Get Thread ID */);
  else
    printf("Hello world from thread %d,
    I am even\n", /* Get Thread ID */);
  }
}
```

Fortran

```fortran
program hello
! Include/Use omp_lib.h/omp_lib ?
implicit none
! Add OMP Directive
if (mod(i,2).eq.1) then
  print *,'Hello from thread',id,', I am
  odd!'
else
  print *,'Hello from thread',id,', I am
  even!'
endif
! End OMP Directive ?
end program hello
```

Exercise 1: Hello World – Solution

C

```
#include <stdio.h>
#include <omp.h>
int main() {
  #pragma omp parallel
  {
  if (id%2==1)
    printf("Hello world from thread %d,
    I am odd\n", omp_get_thread_num());
  else
    printf("Hello world from thread %d,
    I am even\n", omp_get_thread_num());
  }
}
```

Fortran

```
program hello
use omp_lib
implicit none
!$omp parallel
if (mod(i,2).eq.1) then
  print *,'Hello from
thread',omp_get_thread_num(),', Iam
  odd!'
else
  print *,'Hello from thread',
omp_get_thread_num(),', I am
  even!'
endif
!$omp end parallel
end program hello
```

LSU CENTER FOR COMPUTATION & TECHNOLOGY

Working Sharing: Parallel Loops

- Loops are the most likely targets when parallelizing a serial program
  - Syntax:
    - Fortran: `!$omp do`
    - C/C++: `#pragma omp for`
- Other work sharing directives available
  - Sections: `!$omp sections` or `#pragma sections`
  - Tasks: `!$omp task` or `#pragma omp task`
- The parallel and work sharing directive can be combined as
  - `!$omp parallel do` or `#pragma omp parallel for`

Example: Parallel Loops

C	Fortran

C

```
#include <omp.h>

int main() {
int i = 0, n = 100, a[100];
#pragma omp parallel for
for (i = 0; i < n ; i++) {
  a[i] = (i+1) * (i+2);
}
}
```

Fortran

```
program omppardo

use omp_lib
implicit none
integer :: i, n, a(100)

i = 0
n = 100
!$omp parallel
!$omp do
do i = 1, n
  a(i) = i * (i+1)
end do
!$omp end do
!$omp end parallel

end program omppardo
```

Load Balancing (1)

- OpenMP provides different methods to divide iterations among threads,
- indicated by the `schedule` clause
  - Syntax: `schedule (<method>, [chunk size])`
- Methods include
  - `Static`: the default schedule; divide interactions into chunks according to size, then distribute chunks to each thread in a round-robin manner.
  - `Dynamic`: each thread grabs a chunk of iterations, then requests another chunk upon completion of the current one, until all iterations are executed.
  - `Guided`: similar to Dynamic; the only difference is that the chunk size starts large and shrinks to `size` eventually.

Load Balancing (2)

Assuming there are 4 threads working on 100 iterations

	Iterations mapped onto thread			
	0	1	2	3
Static	1-25	26-50	51-75	76-100
Static, 20	1-20, 81-100	21-40	41-60	61-80
Dynamic	1, …	2, …	3, …	4, …
Dynamic, 5	1-5, …	6-10, …	11-15, …	16-20, …

Load Balancing (3)

Schedule	When to use
Static	Even and predictable workload per iteration; scheduling may be done at compilation time, least work at runtime.
Dynamic	Highly variable and unpredictable workload per iteration; most work at runtime.
Guided	Special case of dynamic scheduling; compromise between load balancing and scheduling overhead at runtime

Work Sharing: Section

- Gives a different code block to each thread

C	Fortran
see below	see below

C

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      some_calculation();
    #pragma omp section
      some_more_calculation();
    #pragma omp section
      yet_some_more_calculation();
  }
}
```

Fortran

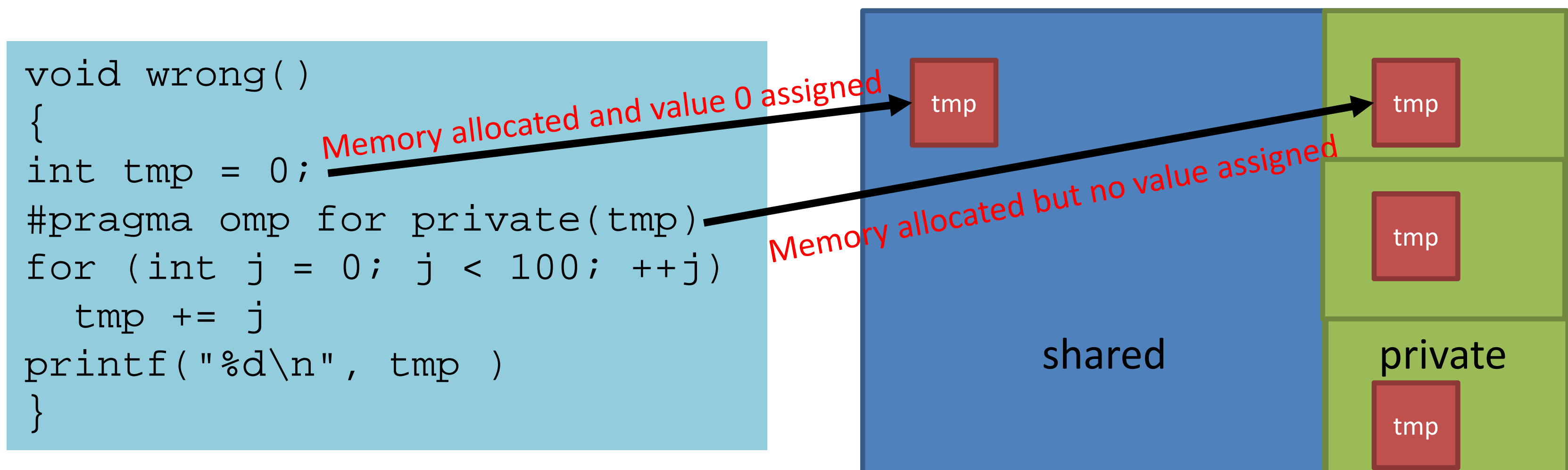
```
!$omp parallel
!$omp sections
!$omp section
  call some_calculation
!$omp section
  call some_more_calculation
!$omp section
  call yet_some_more_calculation
!$omp end sections
!$omp end parallel
```

Scope of Variables

- `Shared(variable list)`
  - Specifies the variables that are shared among all threads
- `Private(variable list)`
  - Creates a local copy of the specified variables for each thread
  - The value is uninitialized!
- `Default(shared|private|none)`
  - Defines the default scope of variables
  - C/C++ API does not have **default(private)**
- Most variables are shared by default
  - A few exceptions: iteration variables; stack variables in subroutines; automatic
- variables within a statement block.

Private Variables

- Not initialized at the beginning of parallel region!
- After parallel region
  - Not defined in OpenMP 2.x
  - 0 in OpenMP 3.x



Special Cases of Private Variables

- `Firstprivate`
  - Initialize each private copy with the corresponding value from the master thread
- `Lastprivate`
  - Allows the value of a private variable to be passed to the shared variable outside the parallel region

```
void wrong()
{
  int tmp = 0;
  #pragma omp for firstprivate(tmp) lastprivate(tmp)
  for (int j = 0; j < 100; ++j)
    tmp += j
  printf("%d\n", tmp )
}
```

Each thread have its copy of tmp set to 0

The shared tmp has the value of the last sequential iteration, i.e. j=99

Pitfalls: False Sharing

- Array elements that are in the same cache line can lead to false sharing.
  - The system handles cache coherence on a cache line basis, not on a byte or word basis.
  - Each update of a single element could invalidate the entire cache line.

```
!$omp parallel
myid = omp_get_thread_num()
nthreads = omp_get_numthreads()
do i = myid+1, n , nthreads
  a(i) = some_function(i)
end do
!$omp end parallel
```

LSU CENTER FOR COMPUTATION & TECHNOLOGY

Pitfalls: Race Condition

- Multiple threads try to write to the same memory location at the same time.
  - Indeterministic results
- Inappropriate scope of variables can cause indeterministic results too.
- When having indeterministic results, set the number of threads to 1 to check
  - If problem persists: scope problem
  - If problem is solved: race condition

```
!$omp parallel do
do i = 1, n
  if (a(i) > max) then
    max = a(i)
  end if
end do
!$omp end parallel do
```

Pitfalls: Hidden Serialization

- Some part of your code could be inherently serial
  - Be aware of library functions which are not transparent to users
  - Example: random number generator – need to use RNGs that are really parallel

Synchronization: Critical and Atomic

- Critical
  - Only one thread at a time can enter a critical region

```
!$omp parallel do
do i = 1, n
  b = some_function(i)
  !$omp critical
  call some_routine(b,x)
end do
!$omp end parallel do
```

- Atomic
  - Only one thread at a time can update a memory location

```
!$omp parallel do
do i = 1, n
  b = some_function(i)
  !$omp atomic
  x = x + b
  end do
!$omp end parallel do
```

Synchronization: Barrier

- "Stop sign" where every thread waits until all threads arrive.
- Purpose: protect access to shared data.
- Syntax:
  - Fortran: `!$omp barrier`
  - C/C++: `#pragma omp barrier`
- A barrier is implied at the end of every parallel region
  - Use the `nowait` clause to turn it off
- Synchronizations are costly so their usage should be minimized!

Reduction

- The reduction clause allows accumulative operations on the value of variables.
  - Syntax: `reduction (operator:variable list)`
- A private copy of each variable which appears in reduction is created as if the private clause is specified.
- Operators
  - Arithmetic: +, -, *
  - Bitwise
  - Logical
  - Intrinsic functions: max, min

Example: Reduction

C	Fortran

C

```c
#include <omp.h>
int main() {
  int i, n = 100, sum , a[100], b[100];
  for (i = 0; i < n; i++) {
    a[i] = i;
    b[i] = 1;
  }
  sum = 0;
  #pragma omp parallel for
    reduction(+:sum)
  for (i = 0; i < n ; i++) {
    sum += a[i] * b[i];
  }
}
```

Fortran

```fortran
program reduction

use omp_lib
implicit none
integer :: i, n, sum , a(100), b(100)
n = 100 ; b = 1; sum = 0

do i = 1 , n
  a(i) = i
end do

!$omp parallel do reduction(+:sum)
do i = 1, n
  sum = sum + a(i) * b(i)
end do
!$omp end parallel do

end program reduction
```

Exercise 2: Calculating Pi

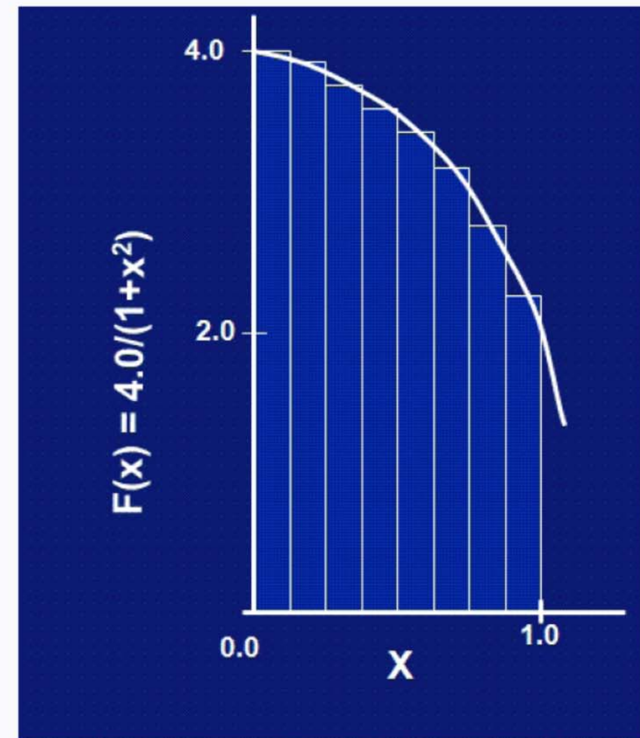
We know that

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

So numerically, we can approxiate pi as the sum of a number of rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Meadows et al, A “hands-on” introduction to OpenMP, SC09



Exercise 2: Serial Programs - C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
  int i;
  long long int n=100000000;
  clock_t start_time, end_time;
  double x, pi;
  double sum = 0.0;
  double step = 1.0/(double) n;
  start_time = clock();
  /* Parallelize the following block of code */
  for (i = 0; i < n; i++) {
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
  end_time = clock();
  printf("pi = %17.15f\n",pi);
  printf("time to compute = %g seconds\n", (double)
   (end_time - start_time)/CLOCKS_PER_SEC);
  return 0;
}
```

Exercise 2: Serial Programs - Fortran

```
program pi_serial

  implicit none
  integer, parameter :: dp=selected_real_kind(14)
  integer :: i
  integer, parameter :: n=100000000
  real(dp) :: x,pi,sum,step,start_time,end_time

  sum = 0d0
  step = 1.d0/float(n)
  call cpu_time(start_time)
  ! Parallelize the following block of code
  do i = 0, n
    x = (i + 0.5d0) * step
    sum = sum + 4.d0 / (1.d0 + x ** 2)
  end do
  pi = step * sum
  call cpu_time(end_time)
  print '(a,f17.15)', "pi = ", pi
  print '(a,f9.6,a)', "time to compute =",end_time
    - start_time, " seconds"

end program pi_serial
```

Exercise 2: Calculating Pi - Solutions

- Serial programs and solutions can be found under `/home/lyan1/traininglab/openmp`

LSU CENTER FOR COMPUTATION & TECHNOLOGY

Target Construct

- New in OpenMP 4
- Purpose: offload data and computation to accelerators (GPU, Xeon Phi, DSP, FPGA etc.)
  - Can specify which device to use, the location and size of data to be transferred to and from the device etc.
- Essential if you plan to develop code on SuperMIC

```
#pragma omp target device(0) map(to: v1,v2) map(from: p)
#pragma omp parallel for
for (i=0; i<N; i++)
  p[i] = v1[i] * v2[i]
```

OpenMP Functions

- Modify/query the number of threads
  - `omp_set_num_threads()`, `omp_get_num_threads()`,
  - `omp_get_thread_num()`, `omp_get_max_threads()`
- Query the number of processors
  - `omp_num_procs()`
- Query whether or not you are in an active parallel region
  - `omp_in_parallel()`
- Control the behavior of dynamic threads
  - `omp_set_dynamic()`,`omp_get_dynamic()`

OpenMP Environment Variables

- `OMP_NUM_THREADS`: set default number of threads to use.
- `OMP_SCHEDULE`: control how iterations are scheduled for parallel loops.

References

- http://www.openmp.org

Questions?