CIS 431/531
Intro to Parallel Computing

Fork-Join Model and OpenMP

Fork-Join

Fundamental way of expressing concurrency within a computation

Fork is called by a thread (parent) to create a new thread (child) of concurrency

- Parent **continues** after the fork operation
- Child begins operation separate from the parent
- Fork **creates** concurrency

Join is called by both the parent and child

- Child joins after it finishes
- Parent waits until child joins
- Join **removes** concurrency

Fork-Join

Fork-join dependency

- Parent must join with its forked children
- Forked children with the same parent can join with parent in **any order**

Fork-join DAG

- What does it look like?

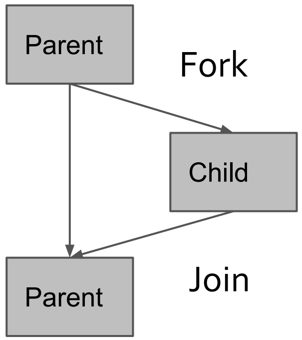
Fork-Join

Fork-join dependency

- Parent must join with its forked children
- Forked children with the same parent can join with parent in **any order**

Fork-join DAG

- What does it look like?



Fork-Join in Unix

Fork-join model comes from basic forms of creating processes and threads in the OS

- Forking a child process from a parent process
  - *fork()* - creates a new child process
  - Process state of parent is **copied** to child process
- Parent process continues to next PC on fork() return
- Child process also starts execution at the next PC
- Parent process can call *waitpid()* for a particular child process
  - If child process has called *join()*, parent continues
  - If child process has not called *join()*, parent blocks/waits

Fork-Join in Unix

Fork-Join “Hello World” in Unix

```c
#include <sys/types.h> /* pid_t */
#include <sys/wait.h>  /* waitpid */
#include <stdio.h>     /* printf, perror */
#include <stdlib.h>    /* exit */
#include <unistd.h>    /* _exit, fork */

int main(void)
{
   pid_t pid;

   pid = fork();

   if (pid == -1) {
      /*
       * When fork() returns -1, an error happened.
       */
      perror("fork failed");
      exit(EXIT_FAILURE);
   }
   else if (pid == 0) {
      /*
       * When fork() returns 0, we are in the child process.
       */
      printf("Hello from the child process!\n");
      _exit(EXIT_SUCCESS);  /* exit() is unreliable here, so _exit must be used */
   }
   else {
      /*
       * When fork() returns a positive number, we are in the parent process
       * and the return value is the PID of the newly created child process.
       */
      int status;
      (void)waitpid(pid, &status, 0);
   }
   return EXIT_SUCCESS;
}
```

POSIX Threads

- Fork-Join in POSIX standard multi-threading interface
  - For general multi-threaded concurrent programming
  - (Largely) independent across implementations/platforms
  - Provides primitives for
    - Thread creation and management
    - Synchronization

POSIX Threads

Thread creation

```
#include <pthread.h>
int pthread_create(
  pthread_t *thread_id,
  const pthread_attr_t *attribute,
  void *(*thread_function)(void *),void *arg);
```

Thread termination

```
void pthread_exit(void *status)
```

Implicitly called when function returns

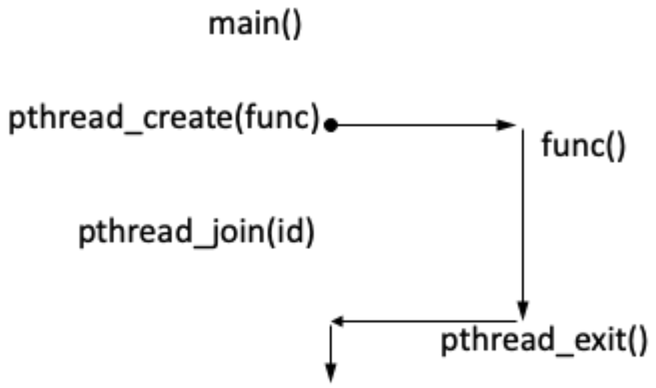
Thread join

```
int pthread_join(
  pthread_t thread_id,
  void **status);
```

POSIX Threads

Example

```
void *func(void *){
   …
}
pthread_t  id;
int  X;
…
pthread_create(&id, NULL, func, &X);
…
pthread_join(id, NULL);
…
```



POSIX Threads

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS     5

void *TaskCode(void *argument)
{
   int tid;

   tid = *((int *) argument);
   printf("Hello World! It's me, thread %d!\n", tid);

   /* optionally: insert more useful stuff here */

   return NULL;
}

int main(void)
{
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int rc, i;

   /* create all threads */
   for (i=0; i<NUM_THREADS; ++i) {
      thread_args[i] = i;
      printf("In main: creating thread %d\n", i);
      rc = pthread_create(&threads[i], NULL, TaskCode, (void *) &thread_args[i]);
      assert(0 == rc);
   }

   /* wait for all threads to complete */
   for (i=0; i<NUM_THREADS; ++i) {
      rc = pthread_join(threads[i], NULL);
      assert(0 == rc);
   }

   exit(EXIT_SUCCESS);
}
```

fork() vs. pthreads

Fork()

- Both parent and child executes the next instruction/PC
- Two identical copies of the address space/code/stack are created

pthreads

- Child thread executes the provided function
- Child thread will **share** open files/signal handlers/working directory with the parent, but get its own stack/registers

Think of the fork as creating an identical copy that executes like the parent, whereas pthread shares data with the parent and operates as an independent worker (doing what the parent tells it to do).

Other Fork-Join Programming Model

Cilk Plus

cilk_spawn B(); // Fork

C();

cilk_sync(); // Join

B() is executed by the child thread

C() is executed by the parent thread

Other Fork-Join Programming Model

OpenMP

Threading Building Blocks (TBB)

OpenACC

Questions?

What is OpenMP?

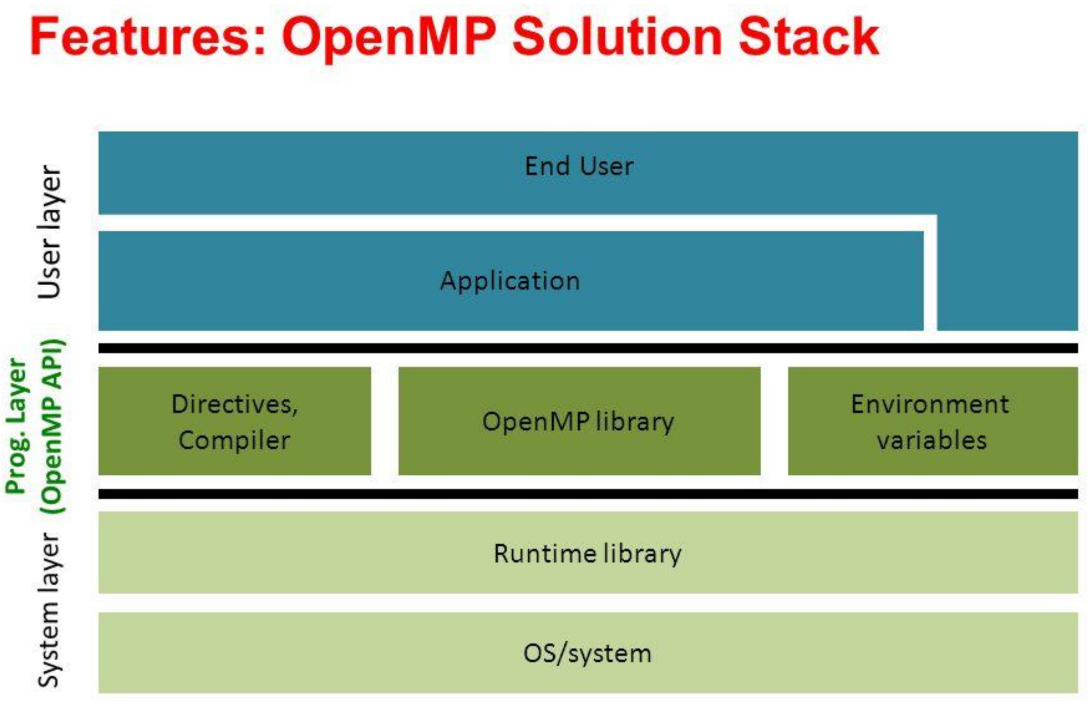
An API for writing multi-threaded (parallel) applications

- Set of compiler directives and library routines
- Greatly simplifies writing multi-threaded code (vs. pthreads)
- Standardizes last 30 years of SMP programming practice

Goals of OpenMP

- Standardized
  - Provide a parallelization standard among a variety of shared memory architectures
  - Defined & endorsed by a number of hardware and software vendors
- Lean
  - Only requires a few lines of directives to parallelize your code
- Easy to use
  - Simple concept (as we will see later)
  - Allows both fine-grained and coarse-grained parallelism
- Portable
  - Supported by most major vendors

OpenMP Stack



OpenMP Features

Designed for multi-processor with shared memory (SMP)

Works with MPI (Message Passing Interface) for distributed system

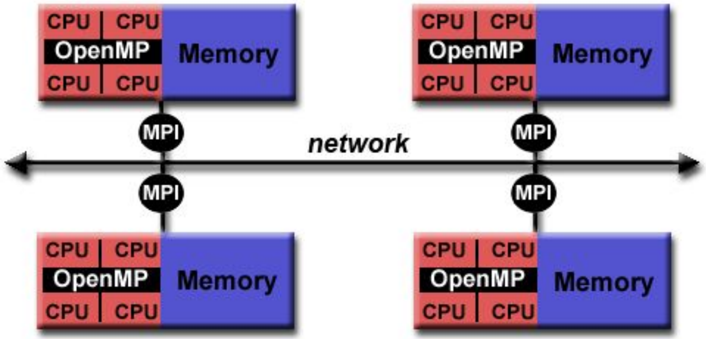
- Hybrid Parallelism (e.g., MPI+X)

Parallelism is achieved through threads

- Thread is the smallest unit of execution (also by the OS)

Explicit Parallelism

- User has full control over parallelization

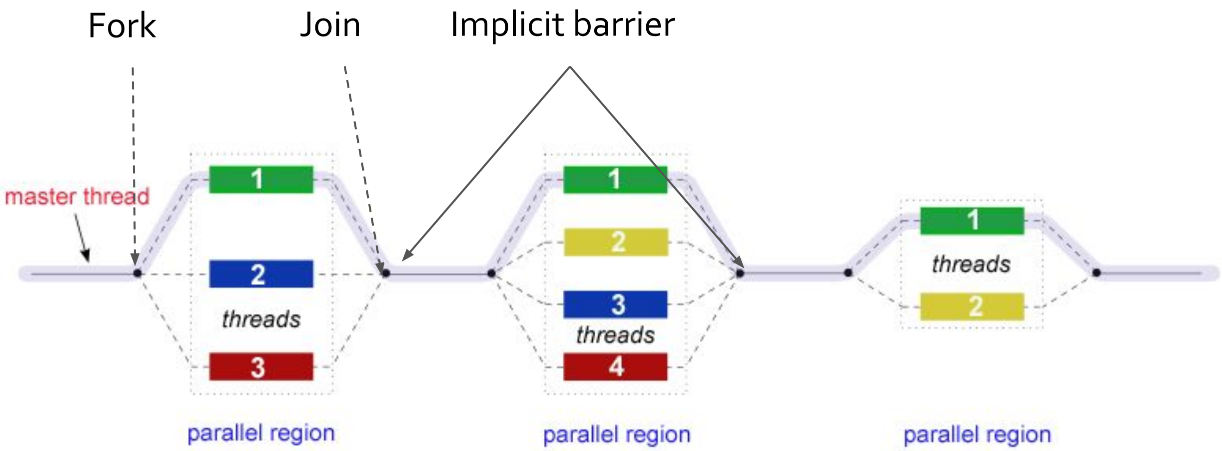


OpenMP Model

Fork-Join Model on OpenMP

- All OpenMP program begins as a **single** process (i.e., the **master thread**)
- Master thread executes alone (sequentially) until a **parallel region** is encountered
    - The program then **forks**
    - Code in the parallel region is executed by multiple threads
    - The thread **joins** when the parallel region is completed
    - The number of parallel regions and threads working on them can be arbitrary
- Within the parallel region
    - Data (e.g., variables) are shared by default
    - Scope of the data can be changed
    - Other parallel regions can exist (nested parallelism)
    - Number of threads can change (depends on vendor support)

OpenMP Model



OpenMP Syntax

Most OpenMP constructs are compiler directives

#pragma omp <directive> [clause …]

```
#pragma omp parallel default(shared) private(a, b)
```

Library Functions

Thread queries (number of threads, thread ID, etc.)

```
int omp_get_num_threads(void)
```

Environment Variables

Setting number of threads, affinity, etc.

```
export OMP_NUM_THREADS=8
```

Why would you want to use environment variables?

Example - Hello World

```
void main()
{




        int ID = 0;
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);

}
```

Example - Hello World

```
#include <omp.h>
void main()
{
#pragma omp parallel
{
    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
}
```

gcc -fopenmp main.c

-qopenmp for Intel compilers (e.g., icc)

-mp for PGI compiler

Example - Hello World

```
#include <omp.h>
void main()
{
#pragma omp parallel
{


    int ID = omp_get_thread_num();
    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
}
```

Example - Hello World

```
#include <omp.h>
void main()
{
int ID;
#pragma omp parallel
{


    ID = omp_get_thread_num();
    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
}
```

- Within the parallel region
  - Data (e.g., variables) are shared by default

Problem?

Example - Hello World

One possible output (the code will behave unpredictably):

- **Race condition** will occur from unintended sharing of variables.
- This is why “join” exists - synchronization to prevent race condition (it will not help in this case).
- However, synchronization is **expensive** - it is best to avoid/minimize synchronization

```
hello(0)  world(3)
hello(3)  world(3)
hello(1)  world(3)
hello(2)  world(3)
```

Example - Hello World

```
#include <omp.h>
void main()
{ int ID = 1;
#pragma omp parallel private(ID)
{


    ID = omp_get_thread_num();
    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
}
```

Example - Hello World

```
#include <omp.h>
void main()
{
    int ID = 1;
#pragma omp parallel private(ID)
{


    ID = omp_get_thread_num();
    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
    printf("%d\n", ID);
}
```

Using pthreads

```
void *perform_work(void *arguments)
{
  int index = *((int *)arguments);
  …
}

  pthread_t threads[NUM_THREADS];
  int thread_args[NUM_THREADS];

  for (i = 0; i < NUM_THREADS; i++) {
    printf("IN MAIN: Creating thread %d.\n", i);
    thread_args[i] = i;
    result_code = pthread_create(&threads[i], NULL, perform_work,
&thread_args[i]);
    assert(!result_code);
  }

  for (i = 0; i < NUM_THREADS; i++) {
    result_code = pthread_join(threads[i], NULL);
    assert(!result_code);
    printf("IN MAIN: Thread %d has ended.\n", i);
  }
```

Work-Sharing Constructs

Divides the work in the code region **between** the threads (vs. all threads executing the entirety of the code region)

Types of work-sharing constructs

- Do/For
- Sections
- Single

Work-Sharing - Do/For

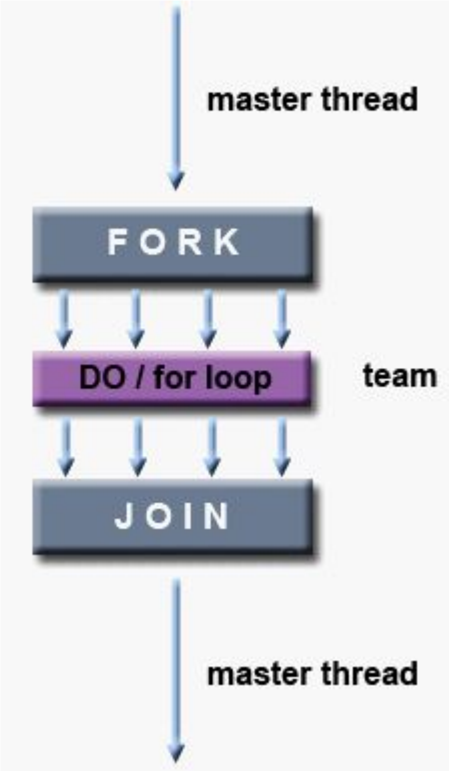
Share iterations of the loop across the threads (i.e., data parallelism)

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < 100; i++) {
        x[i] = 1;
    }
}
```

OR

```
#pragma omp parallel for
for(int i = 0; i < ARR_SIZE; i++) {
        x[i]++;
}
```

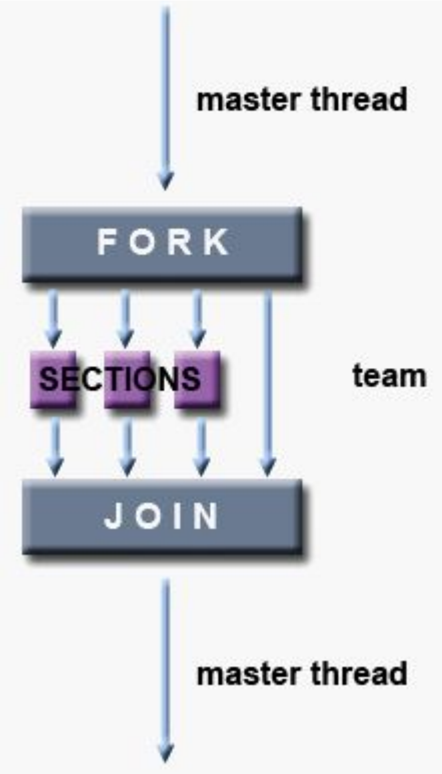
There is an implicit barrier at the end of the loop



Work-Sharing - Sections

Each section can do different parts of the code section (assuming they can be done independently) or completely different work altogether

```
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                for(int i = 0; i < ARR_SIZE/4; i++) {
                    x[i] = 1;
                }
            }
            …
            #pragma omp section
            {
                for(int i = (ARR_SIZE/4)*3; i <
ARR_SIZE; i++) {
                    x[i] = 1;
                }
            }
        }
    }
```



This code has a **similar** effect as using 4 threads with **parallel for**

Work-Sharing - Single

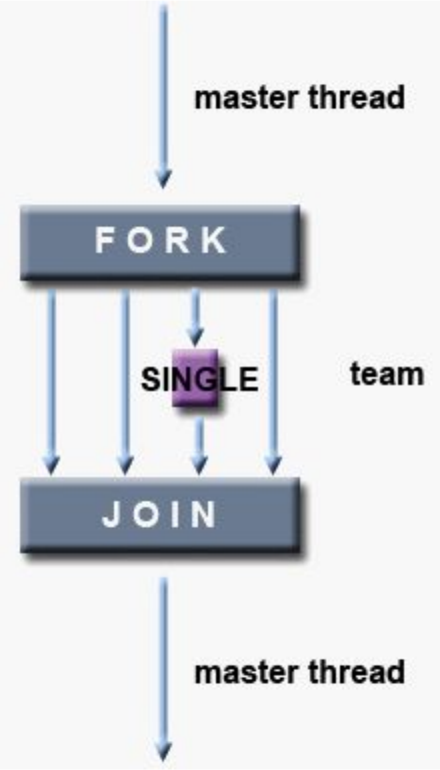
Only 1 thread in the team executes the code section

Why??

- Might be useful when executing code sections that are not thread safe (e.g., IO)

```
#pragma omp parallel
{
    #pragma omp single
    {
        some code…
    }
}
```

Only construct that does **not allow "parallel single"**



Work-Sharing - Master

Only 1 thread in the team executes the code section

- It is the master thread that executes this section and every other thread skips it
- There is no implicit barrier associated with this directive

OpenMP - Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

- **Critical**
- **Atomic**
- Barrier
- Ordered
- Locks

OpenMP - Synchronization

```
int sum = 0;
#pragma omp parallel
{
    sum += omp_get_thread_num();
}
printf("sum = %d\n", sum);
```

What would happen if you ran this with 16 threads?

OpenMP - Synchronization

Critical section

- Mutual exclusion - only one thread at a time can enter the critical region

```
int sum = 0;
#pragma omp parallel
{
    #pragma omp critical
    sum += omp_get_thread_num();
}
printf("sum = %d\n", sum);
```

OpenMP - Synchronization

Atomic variables

- Mutual exclusion - but only to the memory location (i.e., *sum* in this example).

```
int sum = 0;
#pragma omp parallel
{
    #pragma omp atomic
    sum += omp_get_thread_num();
}
printf("sum = %d\n", sum);
```

Questions?

Exercise - Pi

How would you calculate Pi in parallel?

- Hint - What is Pi used for?

Exercise - Pi

How would you calculate Pi in parallel?

- Hint - What is Pi used for? **Calculate the area of a circle**
- In Calculus, what is used to calculate that?

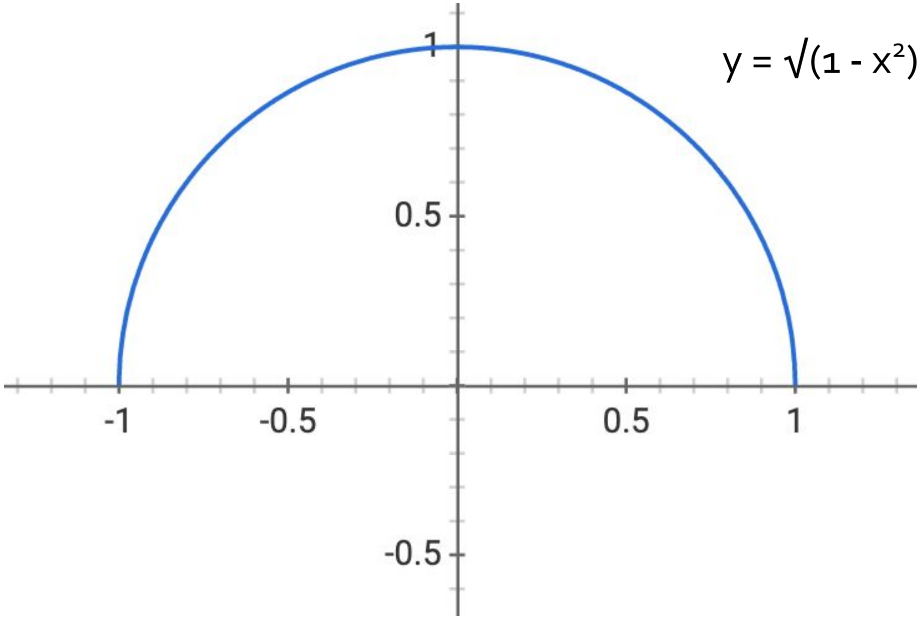
Exercise - Pi

How would you calculate Pi in parallel?

- Hint - What is Pi used for? **Calculate the area of a circle**
- In Calculus, what is used to calculate that? **Integration -> area under a curve**

Exercise - Pi

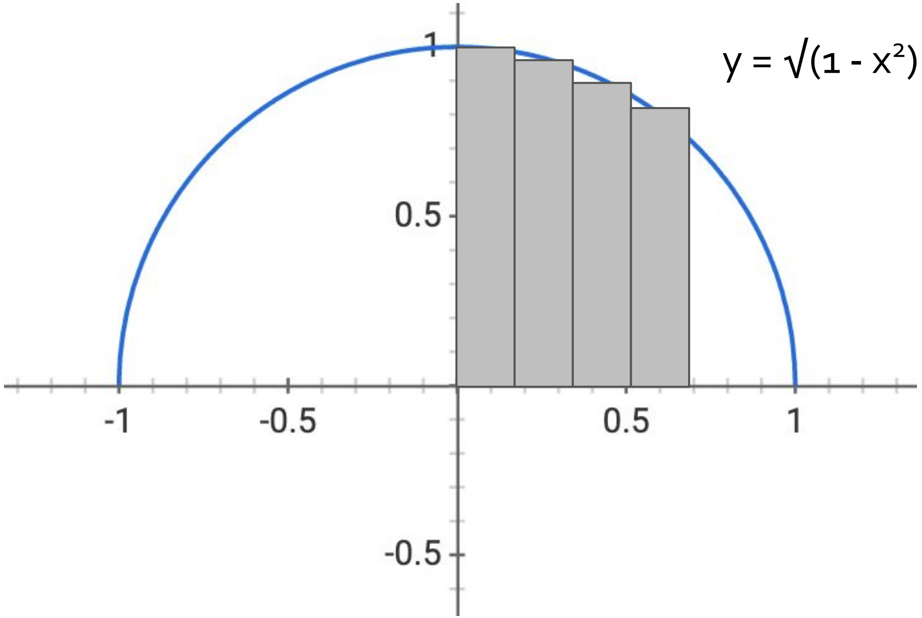
Estimate Pi using an integral of (sqrt(1-x^2)) from -1 to 1



Exercise - Pi

Estimate Pi using an integral of (sqrt(1-x^2)) from -1 to 1

Estimate as sum of rectangular areas

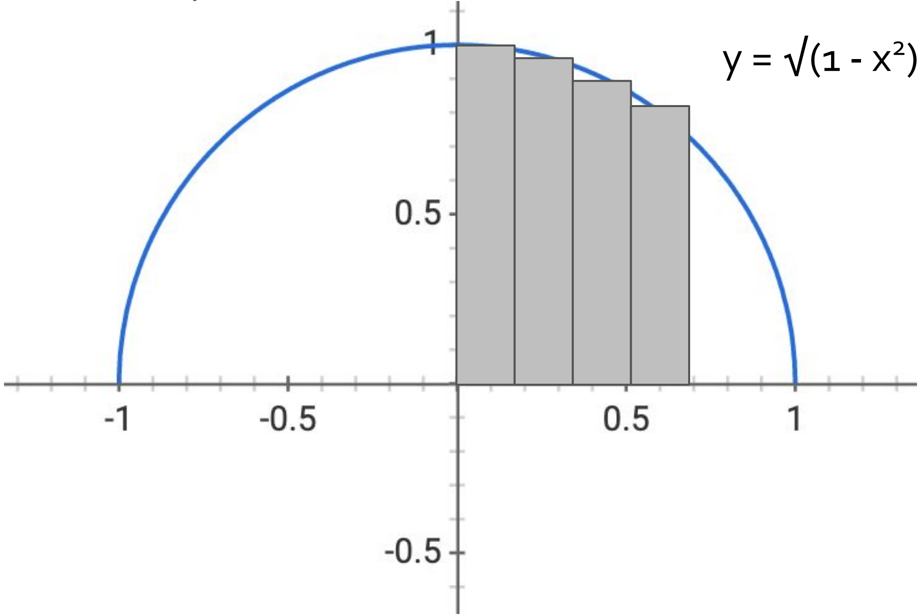


Exercise - Pi

Estimate Pi using an integral of (sqrt(1-x^2)) from -1 to 1

Estimate as sum of rectangular areas

How would you make this more accurate?

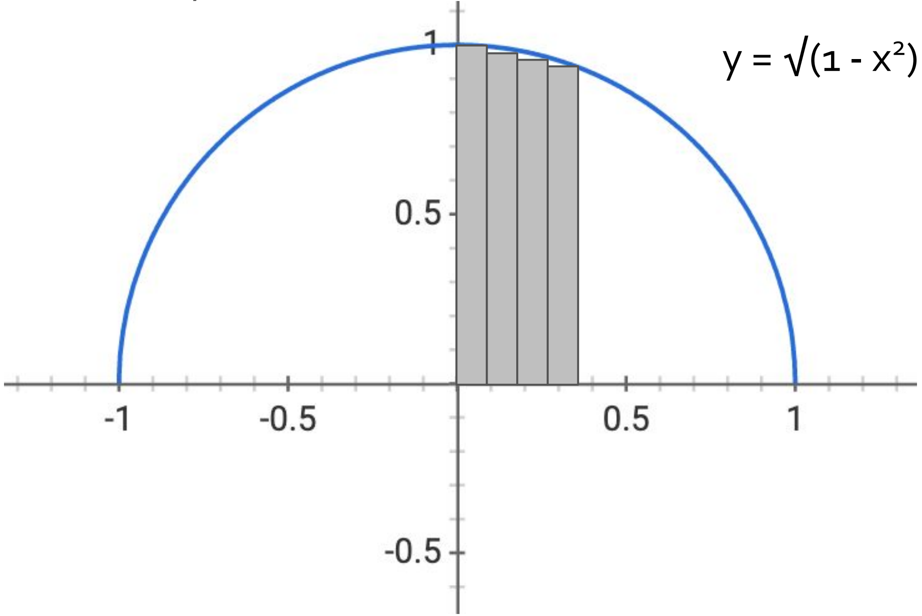


Exercise - Pi

Estimate Pi using an integral of (sqrt(1-x^2)) from -1 to 1

Estimate as sum of rectangular areas

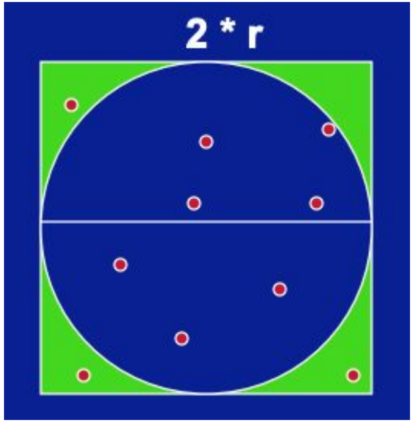
How would you make this more accurate?



Homework - Pi

Use the Monte Carlo Method (i.e., random numbers)

- Throw darts at the square (in green)
- Chance of falling in the circle is proportional to the ratio of areas (circle vs. square)
- Compute Pi by randomly choosing points and counting the fraction that falls in the circle.



Skeleton Code

1. Goto https://bitbucket.org/jeewhanchoi/uoregon-cis431531-f23/src/master/
2. Clone the repo and copy the homework01 directory to your personal repo
3. Read the READ.ME for instructions
4. Do the homework and push the changes to **your** personal repo

Questions?