AMATH 483/583 High Performance Scientific Computing

Lecture 14: OpenMP

Xu Tony Liu, PhD

Paul G. Allen School of Computer Science & Engineering

University of Washington

Seattle, WA

Overview

- Review
  - Summary of C++ features (Parallel)
- Introduction to OpenMP
- OpenMP programming model
- Querying OpenMP environment
- Setting OpenMP environment
- Parallel regions
- Parallel for
- Race conditions
- Reduction

Race condition

- Race condition
  - An unexpected error arises (or can arise) from concurrent accesses to shared variables
- Critical-section problem
  - when n processes/threads all competing to use some shared data, each process/thread has a code segment, called critical section, in which the shared data is accessed
- Race condition solutions
  - Mutual exclusion

Summary of C++ features (Parallel)

- std::thread
  - detach(), join()
- std::mutex, std::lock_guard<T>, std::lock
- std::atomic<T>
- std::async
  - std::future, get(), wait()
  - Async launch strategies
- std::async using lambda function

Two Norm Function (Sequential)

```
double two_norm(const Vector& x) {
  double sum = 0.0;
  for (size_t i = 0; i < x.num_rows(); ++i) {
    sum += x(i) * x(i);
  }
  return std::sqrt(sum);
}
```

Partitioned Vector

```
class PartitionedVector {
public:
  PartitionedVector(size_t M) : num_rows_(M), storage_(num_rows_) {}

        double& operator()(size_t i)       { return storage_[i]; }
  const double& operator()(size_t i) const { return storage_[i]; }

  size_t num_rows() const { return num_rows_; }

  void partition_by_rows(size_t parts) {
    size_t xsize = num_rows_ / parts;
    partitions_.resize(parts+1);
    std::fill(partitions_.begin()+1, partitions_.end(), xsize);
    std::partial_sum(partitions_.begin(), partitions_.end(), partitions_.begin());
  }


private:
  size_t              num_rows_;
  std::vector<double> storage_;
public:
  std::vector<size_t> partitions_;
};
```

Two Norm (Helper Function)

```
double two_norm_part(const PartitionedVector& x, size_t p) {
  double sum = 0.0;
  for (size_t i = x.partitions_[p]; i < x.partitions_[p+1]; ++i) {
    sum += x(i) * x(i);
  }
  return sum;
}

double two_norm_rx(const PartitionedVector& x) {
  std::vector<std::future<double>> futures_;
  for (size_t p = 0; p < x.partitions_.size()-1; ++p) {
    futures_.push_back(std::async(std::launch::async, two_norm_part, std::cref(x), p));
  }

  double sum = 0.0;
  for (size_t p = 0; p < x.partitions_.size()-1; ++p) {
    sum += futures_[p].get();
  }
  return std::sqrt(sum);
}
```

Two Norm (Lambda)

```
double two_norm_l(const PartitionedVector& x) {
  std::vector<std::future<double>> futures_;
  for (size_t p = 0; p < x.partitions_.size()-1; ++p) {
    futures_.emplace_back(std::async(std::launch::async, [&](size_t p) {
    double sum = 0.0;
    for (size_t i = x.partitions_[p]; i < x.partitions_[p+1]; ++i) {
      sum += x(i) * x(i);
    }
    return sum;
  }, p));

  }

  double sum = 0.0;
  for (size_t p = 0; p < x.partitions_.size()-1; ++p) {
    sum += futures_[p].get();
  }
  return std::sqrt(sum);
}
```

Two Norm (Lambda

```
double two_norm(const Vector& x) {
  double sum = 0.0;
  for (size_t i = 0; i < x.num_rows(); ++i) {
    sum += x(i) * x(i);
  }
  return std::sqrt(sum);
}
```

This (straightforward)

Became this (not so straightforward)

```
                rm_l(const Pa
  std::vector<std::future<double>> futures_;
  for (size_t p = 0; p < x.partitions_.size()-1; ++p) {
    futures_.emplace_back(std::async(std::launch::async, [&](size_t p)
    double sum = 0.0;
    for (size_t i = x.partitions_[p]; i < x.partitions_[p+1]; ++i) {
      sum += x(i) * x(i);
    }
    return sum;
  }, p));

  }

  double sum = 0.0;
  for (size_t p = 0; p < x.partitions_.size()-1; ++p) {
    sum += futures_[p].get();
  }
  return std::sqrt(sum);
}
```

Different parts of vector

Finding Concurrency

Partitioned vector

Algorithm Structure

Rewrite algorithm

Would need to do for all

Supporting Structures

PartitionedVector

Fork/join

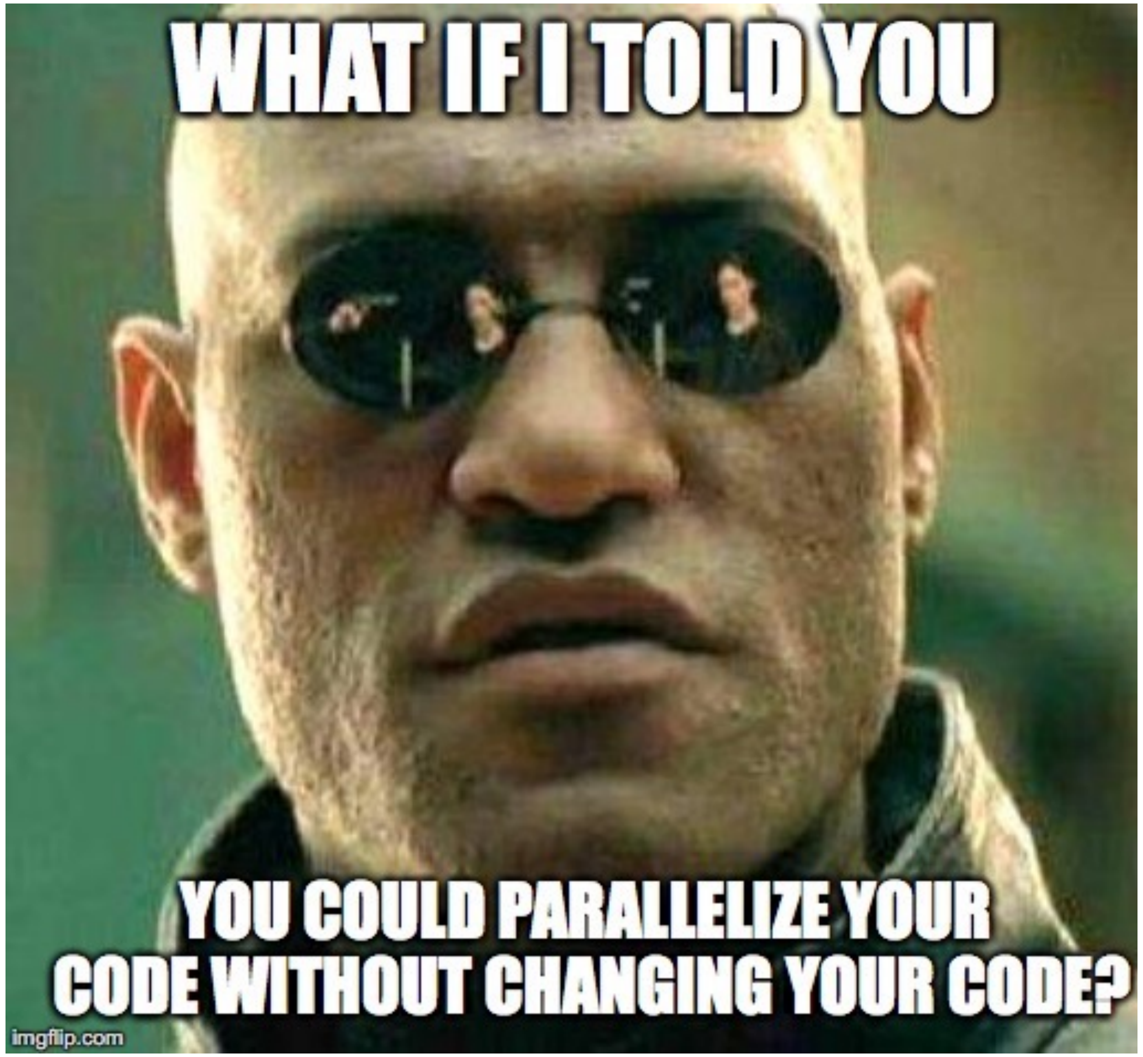
Implementation Mechanisms

C++ tasks/futures

Twice as much code

Different / separate code

What if I told you



```
double two_norm(const Vector& x) {
  double sum = 0.0;
  for (size_t i = 0; i < x.num_rows(); ++i) {
    sum += x(i) * x(i);
  }
  return std::sqrt(sum);
}
```

This does not change

OpenMP™

OpenMP

- **Open M**ulti-**P**rocessing
- Application Program Interface (API) used to explicitly direct ***multi-threaded, shared memory*** parallelism
- Three primary API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables

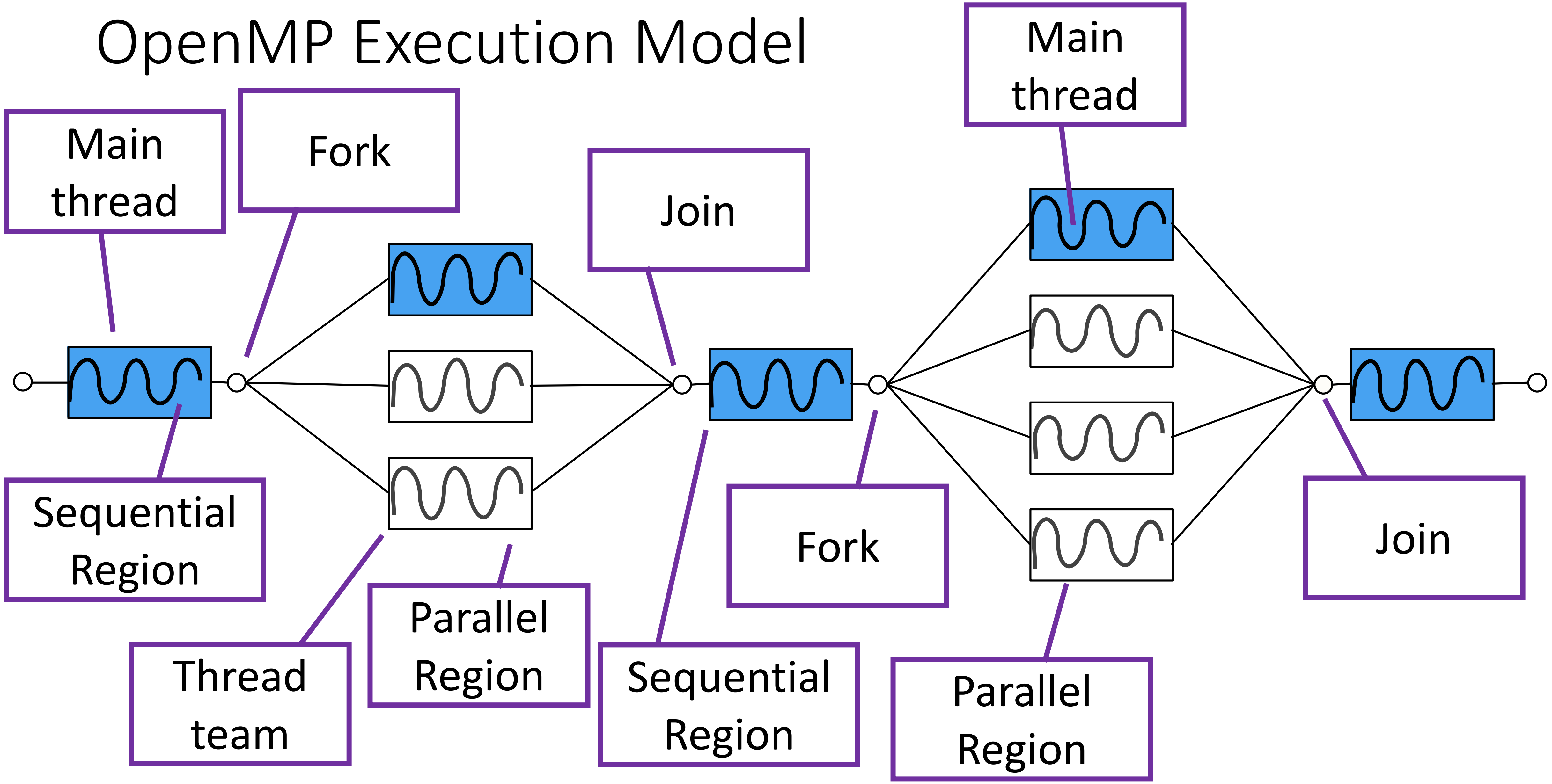
Requires no code changes

Some additions

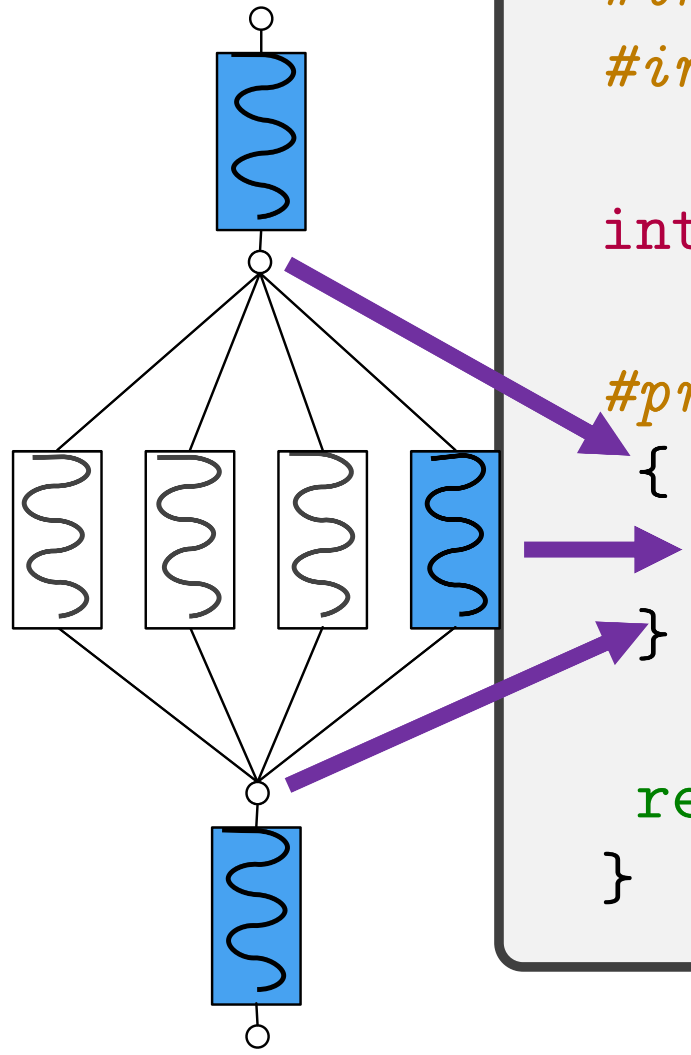
Only for parallel version

Requires no code changes

OpenMP Execution Model



Hello OpenMP v.0



```
#include <iostream>
#include <omp.h>

int main () {

#pragma omp parallel
 {
   std::cout << "Hello OpenMP World!" << std::endl;
 }

 return 0;
}
```

OpenMP Parallel Regions

- A parallel region that is executed by a team consisting of more than one thread

Spawn a team of threads

Each thread will execute the parallel region

```
#include <iostream>
#include <omp.h>

int main () {

#pragma omp parallel
 {
   std::cout << "Hello OpenMP World!" << std::endl;
 }

 return 0;
}
```

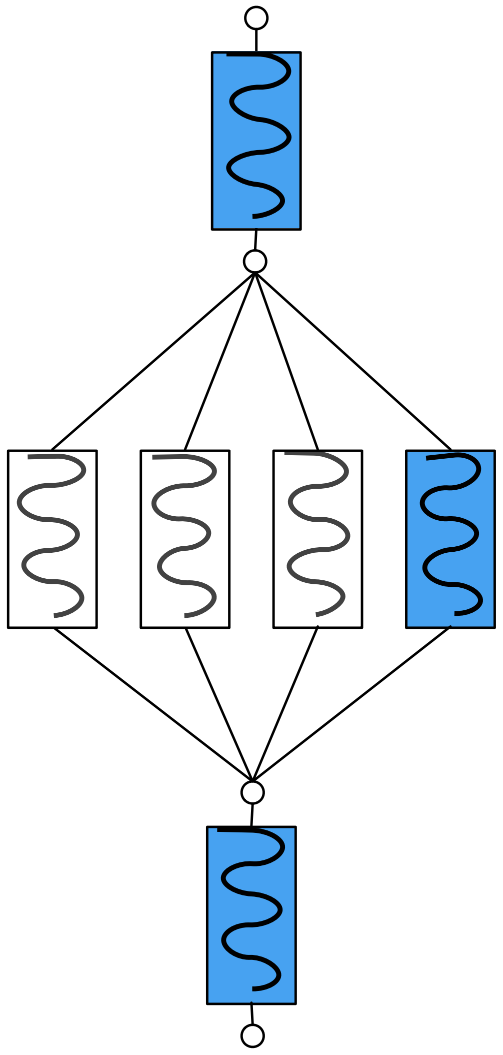
Programming with OpenMP

- How do we start a parallel region?
- How do we end a parallel region?
- What can we do with / in a parallel region?
- Do we need to worry about race conditions? And if so, what do we do about them?
- How do we optimize?
- Do we really not need to change our code?
- What else can we do with OpenMP?
- Example(s)

Querying environment

```
#include <omp.h>

int main(int argc, char* argv[]) {

  size_t numthreads = omp_get_num_threads();
  size_t maxthreads = omp_get_max_threads();
  std::cout << "Number of threads: " << numthreads << std::endl;
  std::cout << "Max threads: " << maxthreads << std::endl;

  return 0;
}
```

Querying the environment



My machine has 8 cores on it

```
$ ./a.out
Number of threads: 1
Max threads: 8
```

Querying environment

- **int omp_get_num_threads(void);**
  - returns the number of threads in the current team

- **int omp_get_max_threads(void);**
  - returns an upper bound on the number of threads that could be used

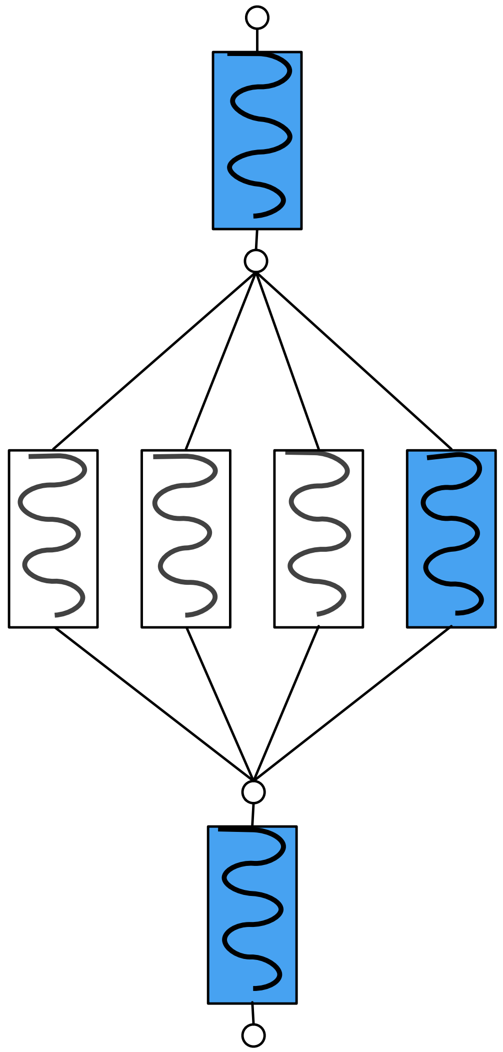
Querying the environment

```
#include <omp.h>

int main(int argc, char* argv[]) {

  size_t maxthreads = omp_get_max_threads();
  std::cout << "Max threads: " << maxthreads << std::endl;
#pragma omp parallel
  {
    size_t numthreads = omp_get_num_threads();
    std::cout << "Number of threads: " + std::to_string(numthreads) + "\n";
  }

  return 0;
}
```

Guess?

Guess?

Querying the environment



```
Max threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
```

Hello OpenMP v.0

```
#include <iostream>
#include <omp.h>

int main () {

#pragma omp parallel
 {
   std::cout << "Hello OpenMP World!" << std::endl;
 }

 return 0;
}
```

```
Hello OpenMP World!Hello
OpenMP World!Hello OpenMP
World!

Hello OpenMP World!Hello
OpenMP World!

Hello OpenMP World!
```

```
Hello OpenMP World!
Hello OpenMP World!
```

Hello OpenMP v.1

```cpp
#include <iostream>
#include <omp.h>

int main () {

#pragma omp parallel
 {
   std::cout << "Hello OpenMP World!\n";
 }

 return 0;
}
```

```
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
```

Hello OpenMP

```
#include <iostream>
#include <omp.h>

int main () {

#pragma omp parallel
 {
   std::cout << "Hello OpenMP World!" << std::endl;
 }

 return 0;
}
```

```
Hello OpenMP World!Hello OpenMP
World!Hello OpenMP World!
Hello OpenMP World!Hello OpenMP World!
Hello OpenMP World!


Hello OpenMP World!
Hello OpenMP World!
```

Explain

```
#include <iostream>
#include <omp.h>

int main () {

#pragma omp parallel
 {
   std::cout << "Hello OpenMP World!\n";
 }

 return 0;
}
```

```
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
Hello OpenMP World!
```

Hello OMP

```
#include <omp.h>

int main () {

#pragma omp parallel
 {
   size_t tid = omp_get_thread_num();
   std::cout << "Hello World from thread = " << tid << std::end

   if (tid == 0) {
     size_t nthreads = omp_get_num_threads();
     std::cout << "Number of threads = " << nthreads << std::en
   }
 }

 return 0;
}
```

Only 1 thread will pass this

Comments?

```
Hello World from thread = Hello
World from thread = Hello
World from thread = Hello
World from thread = Hello
World from thread = 23

Hello World from thread = Hello
World from thread = 7



Hello World from thread = 05

Number of threads = 8


6
4
1
```

Querying environment

- **int omp_get_thread_num(void);**
  - returns the thread number, with in the current team, of the calling thread

- Notice the difference between two
- **int omp_get_num_threads(void);**
  - returns the number of threads in the current team

Do NOT confuse these two

Hello OMP

```
#include <omp.h>

int main () {

#pragma omp parallel
 {
   size_t tid = omp_get_thread_num();
   std::cout << "Hello World from thread = " << tid << std::endl;

   if (tid == 0) {
     size_t nthreads = omp_get_num_threads();
     std::cout << "Number of threads = " << nthreads << std::endl
   }
 }

 return 0;
}
```

How to modify this?

To get this output?

```
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 4
Hello World from thread = 1
Hello World from thread = 5
Hello World from thread = 3
Hello World from thread = 6
```

Setting the environment

```
#include <omp.h>

int main(int argc, char* argv[]) {
  omp_set_num_threads(std::stoi(argv[1]));
  size_t maxthreads = omp_get_max_threads();
  std::cout << "Max threads: " << maxthreads << std::endl;
#pragma omp parallel
  {
    size_t numthreads = omp_get_num_threads();
    std::cout << "Number of threads: " + std::to_string(numthreads) + "\n";
  }

  return 0;
}
```

Guess?

Guess?

Setting the environment

```cpp
#include <omp.h>

int main(int argc, char* argv[]) {
  omp_set_num_threads(std::stoi(argv[1]));
  size_t maxthreads = omp_get_max_threads();
  std::cout << "Max threads: " << maxthreads << std:
#pragma omp parallel
  {
    size_t numthreads = omp_get_num_threads();
    std::cout << "Number of threads: " + std::to_str
  }

  return 0;
}
```

```
$./a.out 1
Max threads: 1
Number of threads: 1

$./a.out 2
Max threads: 2
Number of threads: 2
Number of threads: 2
```

Setting environment

- **void omp_set_num_threads(int** num_threads**);**
  - affects the number of threads to be used for subsequent parallel regions

- An OpenMP Environment Variable - OMP_NUM_THREADS
  - set the number of threads using the environment variable OMP_NUM_THREADS

  ```
  $ export OMP_NUM_THREADS=<number of threads to use>
  ```

  - query the number of threads using the environment variable OMP_NUM_THREADS

  ```
  $ echo $OMP_NUM_THREADS
  ```

Setting the environment

```
#include <omp.h>

int main(int argc, char* argv[]) {

  size_t maxthreads = omp_get_max_threads();
  std::cout << "Max threads: " << maxthreads << std::endl;
#pragma omp parallel
  {
    size_t numthreads = omp_get_num_threads();
    std::cout << "Number of threads: " + std::to_string(numthreads) + "\n";
  }

  return 0;
}
```

Setting the environment

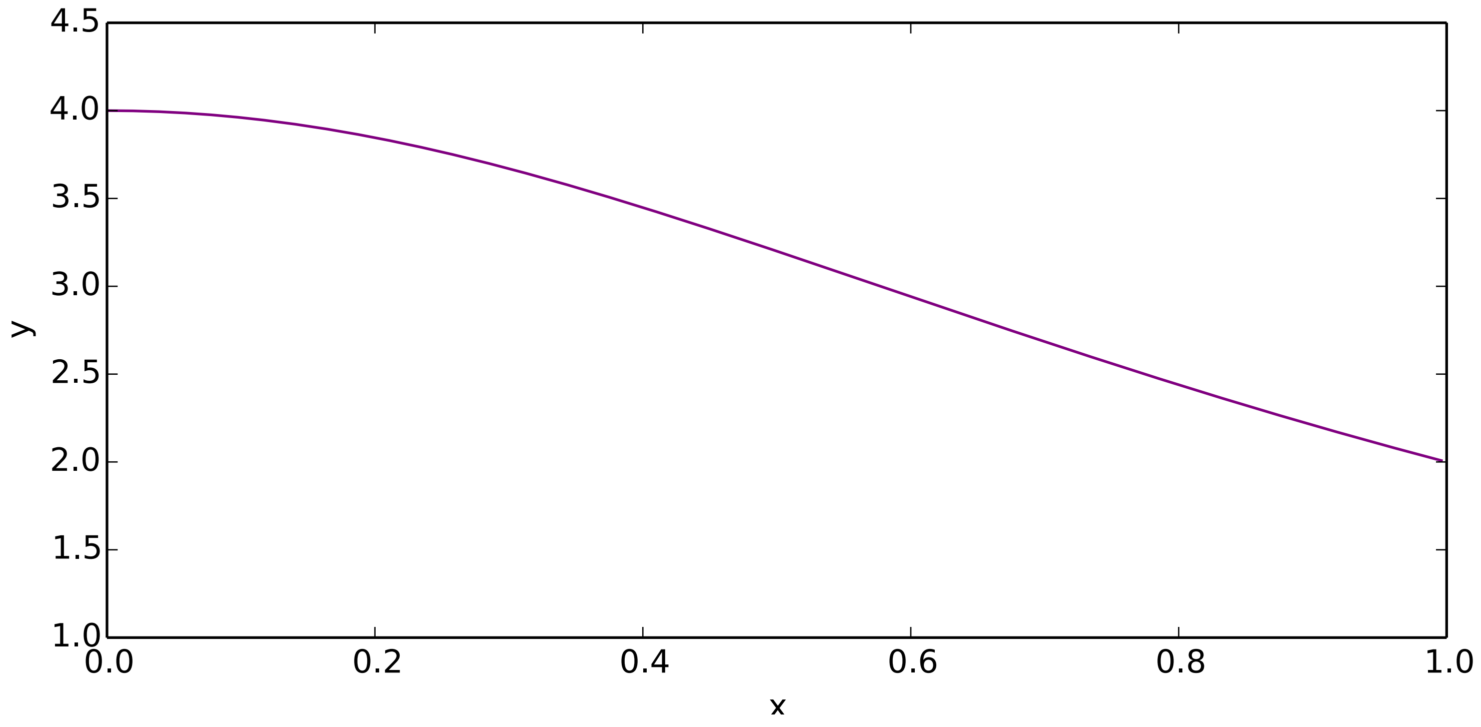
```
$ export OMP_NUM_THREADS=1; ./a.out
Max threads: 1
Number of threads: 1

$ export OMP_NUM_THREADS=2; ./a.out
Max threads: 2
Number of threads: 2
Number of threads: 2
```

Example

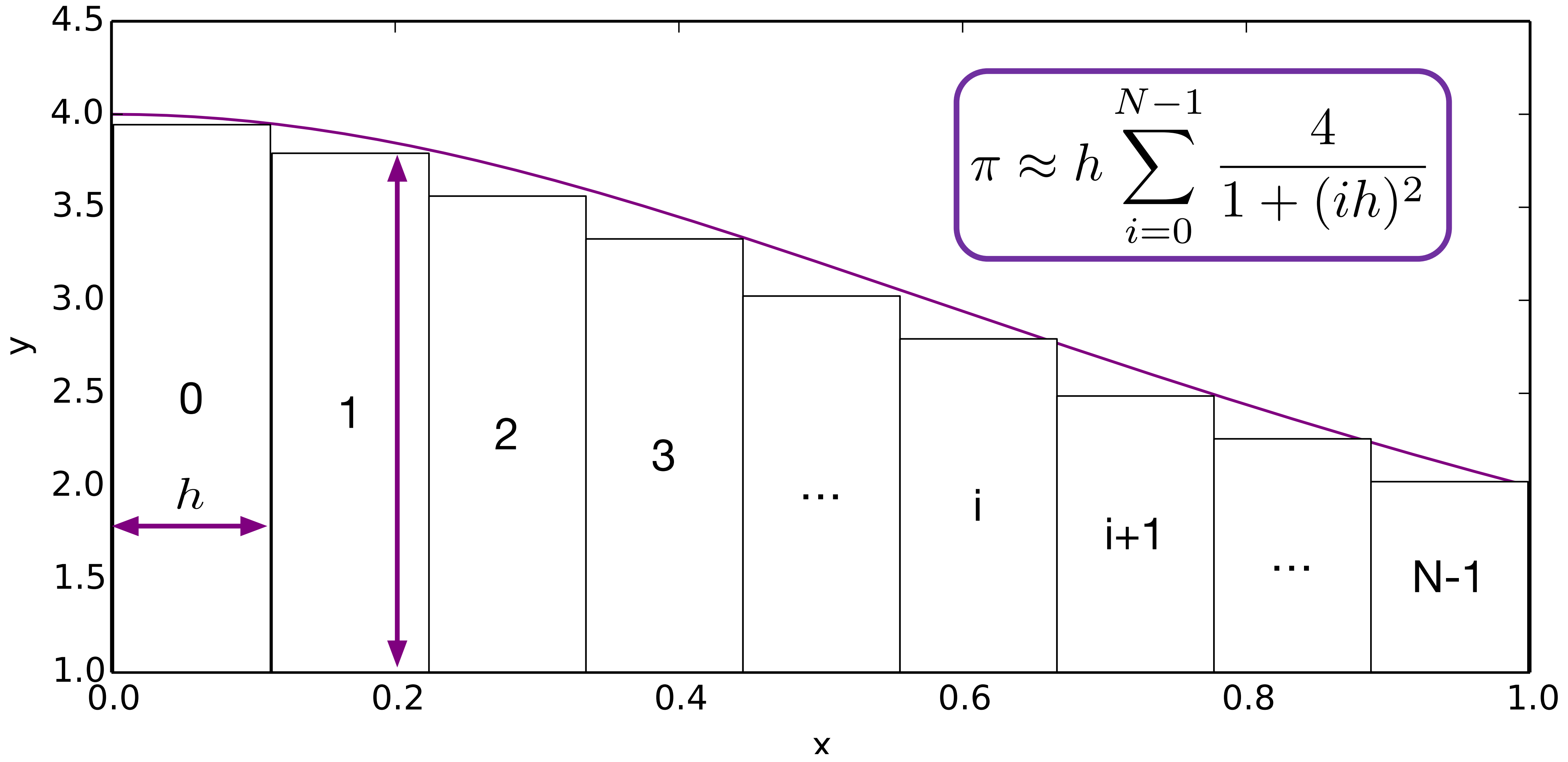
- Find the value of π

- Using formula

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

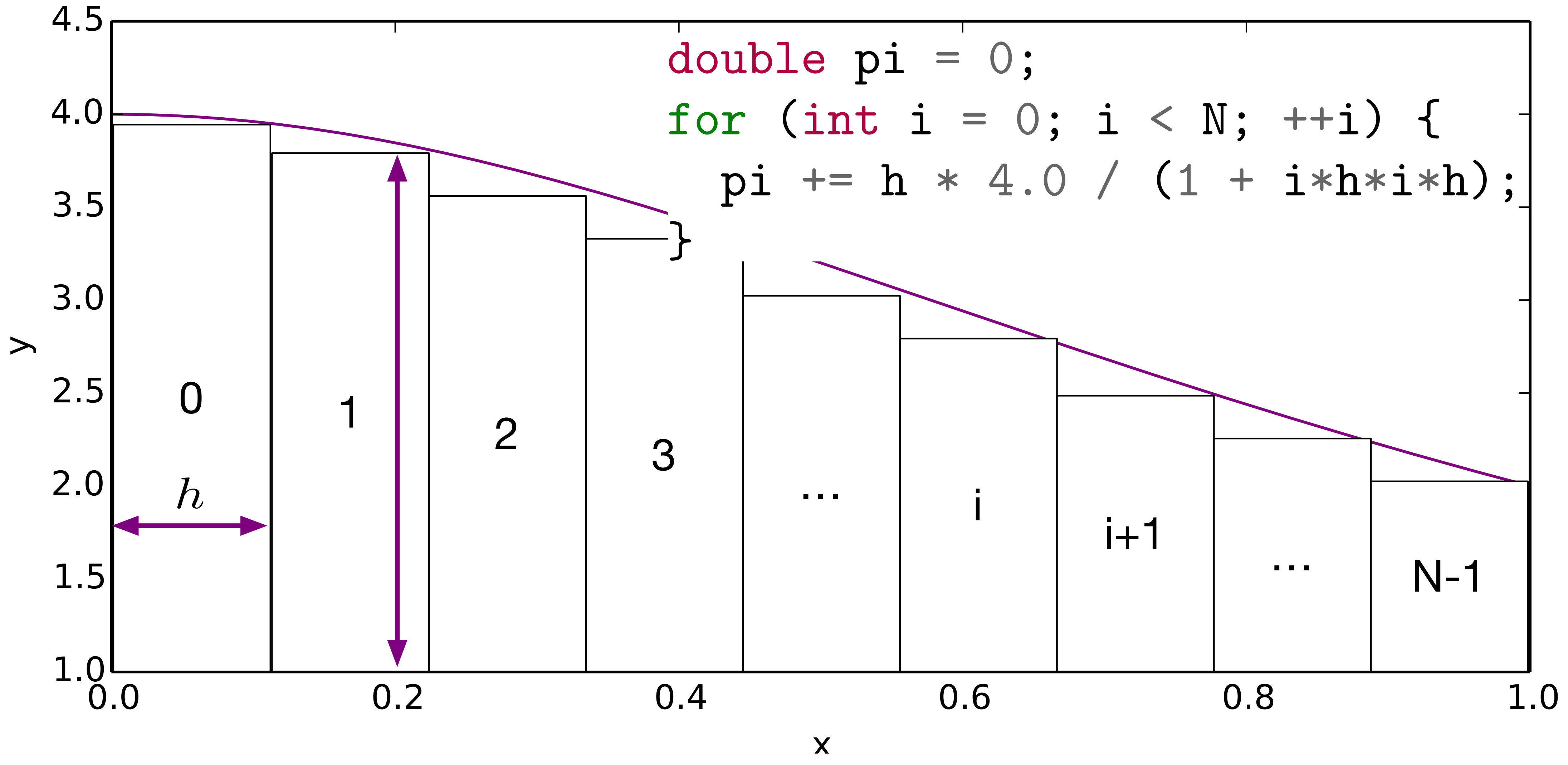


Numerical Quadrature



$$\pi \approx h \sum_{i=0}^{N-1} \frac{4}{1+(ih)^2}$$

Numerical Quadrature



```
double pi = 0;
for (int i = 0; i < N; ++i) {
  pi += h * 4.0 / (1 + i*h*i*h);
}
```

OMP pi 1

```
int main(int argc, char* argv[]) {
  size_t intervals          = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                  = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
↪  std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

Guess?

Output

```
==== pi_omp_1 ====
pi is approximately 3.14159265858898
error is 4.99919039498309e-09
0.740u 0.000s 0:00.75 98.6%     0+0k 32+0io 0pf+0w

pi is approximately 3.14173777858892
error is 0.000145124999126889
4.990u 0.000s 0:02.50 199.6%    0+0k 32+0i

pi is approximately 3.14159265858898
error is 4.99919039498309e-09
27.130u 0.000s 0:06.82 397.8%   0+0k 32+0io 0pf+0w
```

Running with 1 thread

Running with 2 thread

Running with 4 thread

#pragma omp parallel

- #pragma omp parallel spawns a team of threads

Spawn a team of threads

Each thread will execute the parallel region

```
int main(int argc, char* argv[]) {
  size_t intervals            = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                    = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
↪   std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

What Happened?

Shared variable?

Race Condition?

```
int main(int argc, char* argv[]) {
  size_t intervals          = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                  = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
 ↪  std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

Before

```
int main(int argc, char* argv[]) {
  size_t intervals           = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                   = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
↪   std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

After

```
int main(int argc, char* argv[]) {
  size_t intervals         = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                 = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel for
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
↪  std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

Shared variable

Race condition

Guess?

Output

```
==== pi_omp_2 ====
pi is approximately 3.14159265858898
error is 4.99919039498309e-09
0.740u 0.000s 0:00.74 100.0%    0+0k 32+0io 0pf+0w

pi is approximately 1.85459043800302
error is 1.28700221558677
2.520u 0.000s 0:01.26 200.0%    0+0k 32+0io 0pf+0w

pi is approximately 0.62208122839478
error is 2.51951142519501
6.220u 0.020s 0:01.61 387.5%    0+0k 32+0io 0pf+0w
```

Running with 1 thread

Running with 2 thread

Running with 4 thread

#pragma omp parallel for

- #pragma omp for divides loop iterations between the spawned threads

Divides loop iterations between the spawned threads

```
int main(int argc, char* argv[]) {
  size_t intervals            = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                    = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel for
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
↪   std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

#pragma omp parallel for

spawn threads

```
#pragma omp parallel for
for(int i = 1; i < 100; ++i)
{
 ...
}
```

Spawn threads, then divides loop iterations between the spawned threads

Equivalent

```
#pragma omp parallel
{
        #pragma omp for
        for(int i = 1; i < 100; ++i)
        {
        ...
        }
}
```

Divides loop iterations between the spawned threads

Before

```
int main(int argc, char* argv[]) {
  size_t intervals           = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                   = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel for
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
 ↪  std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

After

```
int main(int argc, char* argv[]) {
  size_t intervals         = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                 = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel reduction(+:pi)
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
↪  std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

Guess?

Output

```
==== pi_omp_3 ====
pi is approximately 3.14159265858898
error is 4.99919039498309e-09
0.740u 0.000s 0:00.75 98.6%     0+0k 32+0io 0pf+0w

pi is approximately 6.28318531717797
error is 3.14159266358817
1.500u 0.000s 0:00.75 200.0%    0+0k 32+0io 0pf+0w

pi is approximately 12.5663706343559
error is 9.42477798076614
3.130u 0.020s 0:00.79 398.7%    0+0k 32+0io 0pf+0w
```

Running with 1 thread

Running with 2 thread

Running with 4 thread

#pragma omp reduction(+:pi)

Syntax:

reduction([reduction-modifier**,**]reduction-identifier **:** list**)**

- Perform some forms of recurrence calculations in parallel
- *reduction-modifier* (optional) is one of the following:
  - inscan/task/default

Too advanced for now

- A *reduction-identifier* is either an *id-expression* or one of the following operators: +, -, *, &, |, ^, && and ||.
- *List* could have multiple list items
- The type of each list item must be valid for the *reduction-identifier.*
- For each list item, a private copy is created in each task/thread.
- At the end of the region, the original list item is updated with the values of the private copies using the combiner associated with the *reduction-identifier*.

#pragma omp reduction(+:pi)

```
int main(int argc, char* argv[]) {
  size_t intervals           = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                   = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel reduction(+:pi)
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
↪  std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

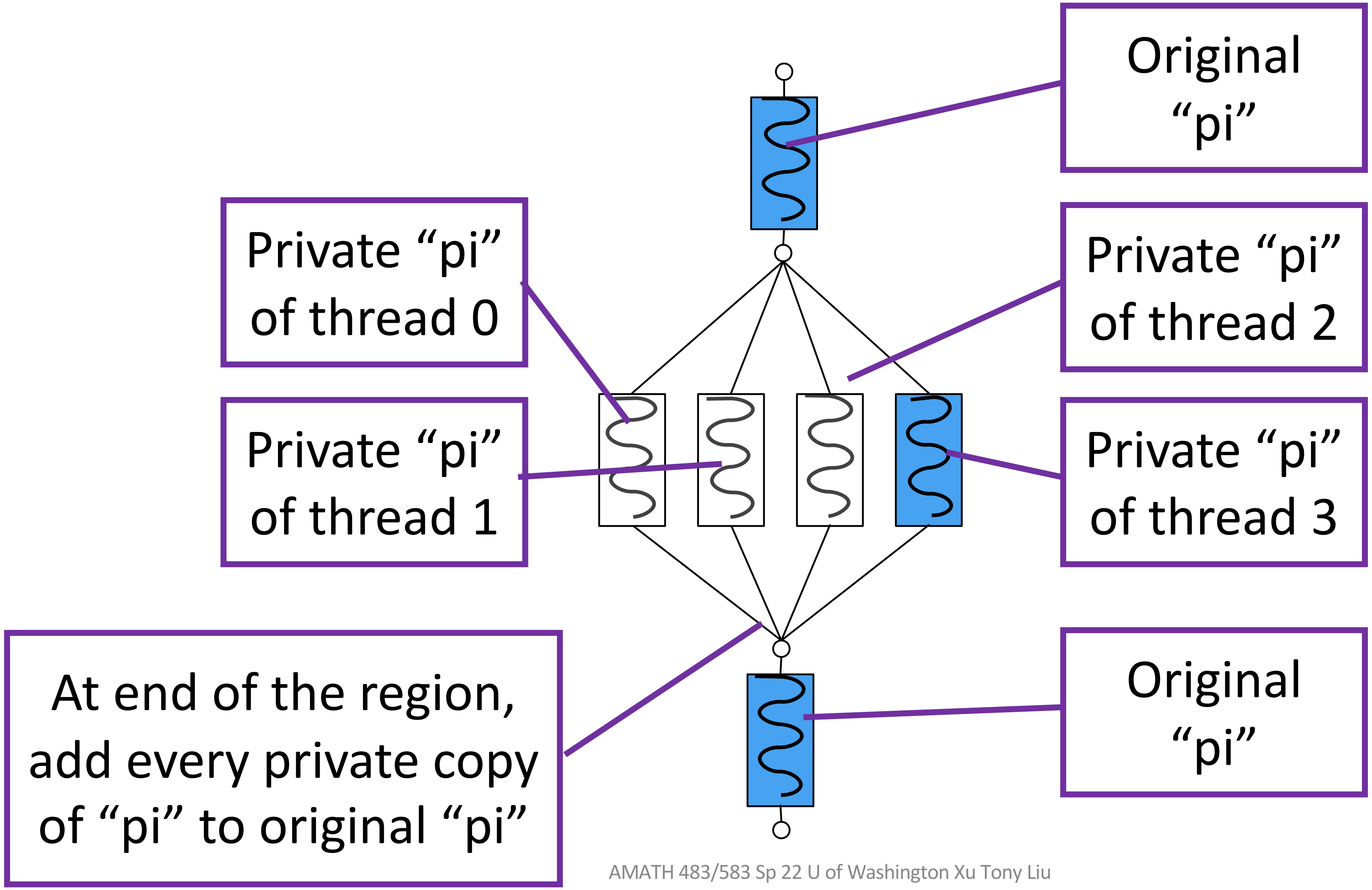
reduction-identifier is "+"

list item is variable "pi"

Meaning?

Add every private copy of "pi" up in parallel

#pragma omp reduction(+:pi)



Before

```
int main(int argc, char* argv[]) {
  size_t intervals           = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                   = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel reduction(+:pi)
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
↪  std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

After

```
int main(int argc, char* argv[]) {
  size_t intervals           = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                   = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel for reduction(+:pi)
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi <<
↪   std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

Spawn a team of threads

Each thread will execute the parallel region

Divides loop iterations between the spawned threads

Add every private copy of “pi” up in parallel

Output

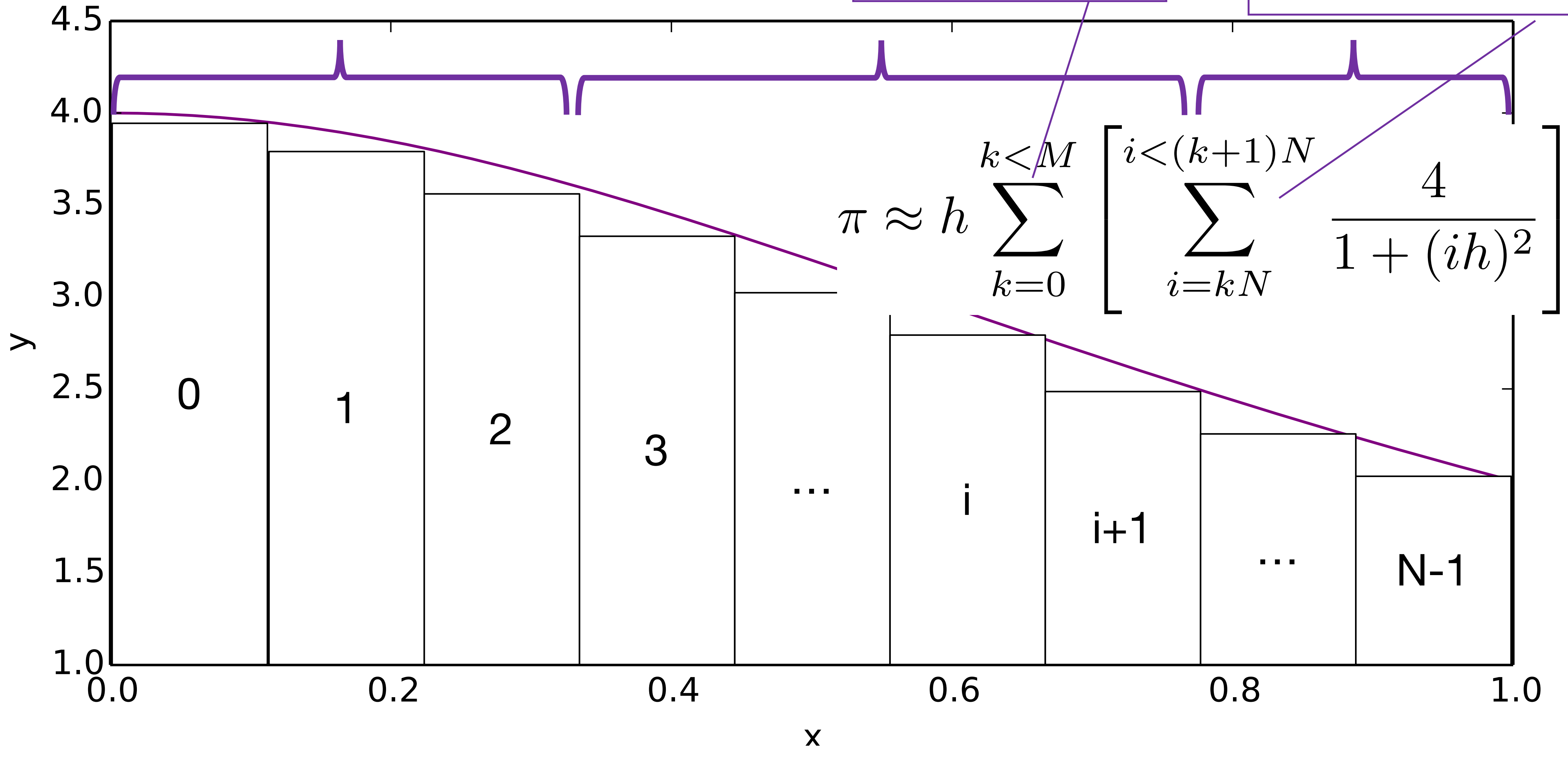
```
==== pi_omp_4 ====
pi is approximately 3.14159265858898
error is 4.99919039498309e-09
0.740u 0.000s 0:00.74 100.0%     0+0k 32+0io 0pf+0w

pi is approximately 3.14159265858936
error is 4.99956342991936e-09
0.750u 0.000s 0:00.38 197.3%     0+0k 32+0i

pi is approximately 3.14159265859013
error is 5.00033436878766e-09
0.850u 0.000s 0:00.24 354.1%     0+0k 32+0io 0pf+0w
```

Running with
1 thread

Running with
2 thread

Running with
4 thread

Finding Concurrency



Sum over partial sums

Partial sum

$$\pi \approx h \sum_{k=0}^{k<M} \left[\sum_{i=kN}^{i<(k+1)N} \frac{4}{1+(ih)^2} \right]$$

Sequential Implementation (Two Nested Loops)

```
  double h = 1.0 / (double) intervals;

double pi = 0.0;
for (int k = 0; k < intervals; k += blocksize) {
  double partial_pi = 0.0;
  for (int i = k; i < (k+blocksize); ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  pi += h * partial_pi;
}
```

Discretization

For each set of discretized points

Compute partial sum

Accumulate final sum

Sequential v.0

```
size_t blocksize = num_intervals / num_blocks;
double h = 1.0 / (double) num_intervals;
double pi = 0.0;
for (size_t k = 0; k < num_intervals; k += blocksize)
{
  double partial_pi = 0.0;
  for (size_t i = k; i < (k+blocksize); ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  pi += h * partial_pi;
}
```

Sequential v.0.5

```
double h = 1.0 / (double) num_intervals;
double pi = 0.0;
for (size_t k = 0; k < num_blocks; ++k)
{
  double partial_pi = 0.0;
  for (size_t i = k; i < num_intervals; i += num_blocks) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  pi += h * partial_pi;
}
```

Sequential v.1

```
size_t blocksize = num_intervals / num_blocks;
double h = 1.0 / (double) num_intervals;
double pi = 0.0;
for (size_t k = 0; k < num_blocks; ++k)
{
  size_t begin = k * blocksize;
  size_t end   = (k + 1) * blocksize;

  double partial_pi = 0.0;
  for (size_t i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  pi += h * partial_pi;
}
```

Sequential v.2

```
size_t blocksize = num_intervals / num_blocks;
double h = 1.0 / (double) num_intervals;
double pi = 0.0;
for (size_t k = 0; k < num_blocks; ++k)
{
  size_t tid   = k;
  size_t begin = tid * blocksize;
  size_t end   = (tid + 1) * blocksize;

  double partial_pi = 0.0;
  for (size_t i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  pi += h * partial_pi;
}
```

Sequential v.3

```
double partial_pi(size_t k, double h, size_t blocksize)
{
  size_t tid   = k;
  size_t begin = tid * blocksize;
  size_t end   = (tid + 1) * blocksize;

  double partial_pi = 0.0;
  for (size_t i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  return partial_pi;
}

  size_t blocksize = num_intervals / num_blocks;
  double h = 1.0 / (double) num_intervals;
  double pi = 0.0;
  for (size_t k = 0; k < num_blocks; ++k)  {
    pi += h * partial_pi(k, h, blocksize);
  }
```

Task version

```
double partial_pi(size_t k, double h, size_t blocksize)
{
  size_t tid   = k;
  size_t begin = tid * blocksize;
  size_t end   = (tid + 1) * blocksize;

  double partial_pi = 0.0;
  for (size_t i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  return partial_pi;
}


  size_t blocksize = num_intervals / num_blocks;
  double h = 1.0 / (double) num_intervals;
  double pi = 0.0;

  std::vector<std::future<double>> futures;
  for (size_t k = 0; k < num_blocks; ++k)  {
    futures.push_back(std::async(std::launch::async, partial_pi, k, h, blocksize));
  }

  for (size_t k = 0; k < num_blocks; ++k)  {
    pi += h * futures[k].get();
  }
```

Sequential

```
size_t blocksize = num_intervals / num_blocks;
double h = 1.0 / (double) num_intervals;
double pi = 0.0;
for (size_t k = 0; k < num_blocks; ++k)
{
  size_t tid = k;
  size_t begin = tid * blocksize;
  size_t end   = (tid + 1) * blocksize;

  double partial_pi = 0.0;
  for (unsigned long i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i * h * i * h));
  }
  pi += h* partial_pi;
}
```

Before

```
size_t blocksize = num_intervals / num_blocks;
double h = 1.0 / (double) num_intervals;
double pi = 0.0;
for (size_t k = 0; k < num_blocks; ++k)
{
  size_t tid = k;
  size_t begin = tid * blocksize;
  size_t end   = (tid + 1) * blocksize;

  double partial_pi = 0.0;
  for (unsigned long i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i * h * i * h));
  }
  pi += h* partial_pi;
}
```

After

```
size_t blocksize = num_intervals / num_blocks;
double h = 1.0 / (double) num_intervals;
double pi = 0.0;
#pragma omp parallel
{
  size_t tid = omp_get_thread_num();
  size_t begin = tid * blocksize;
  size_t end   = (tid + 1) * blocksize;

  double partial_pi = 0.0;
  for (unsigned long i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i * h * i * h));
  }
  pi += h* partial_pi;
}
```

Shared variable

Add reduction over ”pi” with “+” *reduction-identifier*!

Race condition

Solution?

Two Norm Function (Sequential)

- How to parallelize two_norm using OpenMP?

```
double two_norm(const Vector& x) {
  double sum = 0.0;
  for (size_t i = 0; i < x.num_rows(); ++i) {
    sum += x(i) * x(i);
  }
  return std::sqrt(sum);
}
```

Two Norm Function (OpenMP)

Add parallel region to spawn a team of threads

Now each thread will execute the parallel region

```
double two_norm(const Vector& x) {
  double sum = 0.0;
  #pragma omp parallel
  for (size_t i = 0; i < x.num_rows(); ++i) {
    sum += x(i) * x(i);
  }
  return std::sqrt(sum);
}
```

However, every thread is doing the same loop iterations!

Two Norm Function (OpenMP)

Shared variable

Divides loop iterations between the spawned threads

Race condition

```
double two_norm(const Vector& x) {
  double sum = 0.0;
  #pragma omp parallel for
  for (size_t i = 0; i < x.num_rows(); ++i) {
    sum += x(i) * x(i);
  }
  return std::sqrt(sum);
}
```

Two Norm Function (OpenMP)

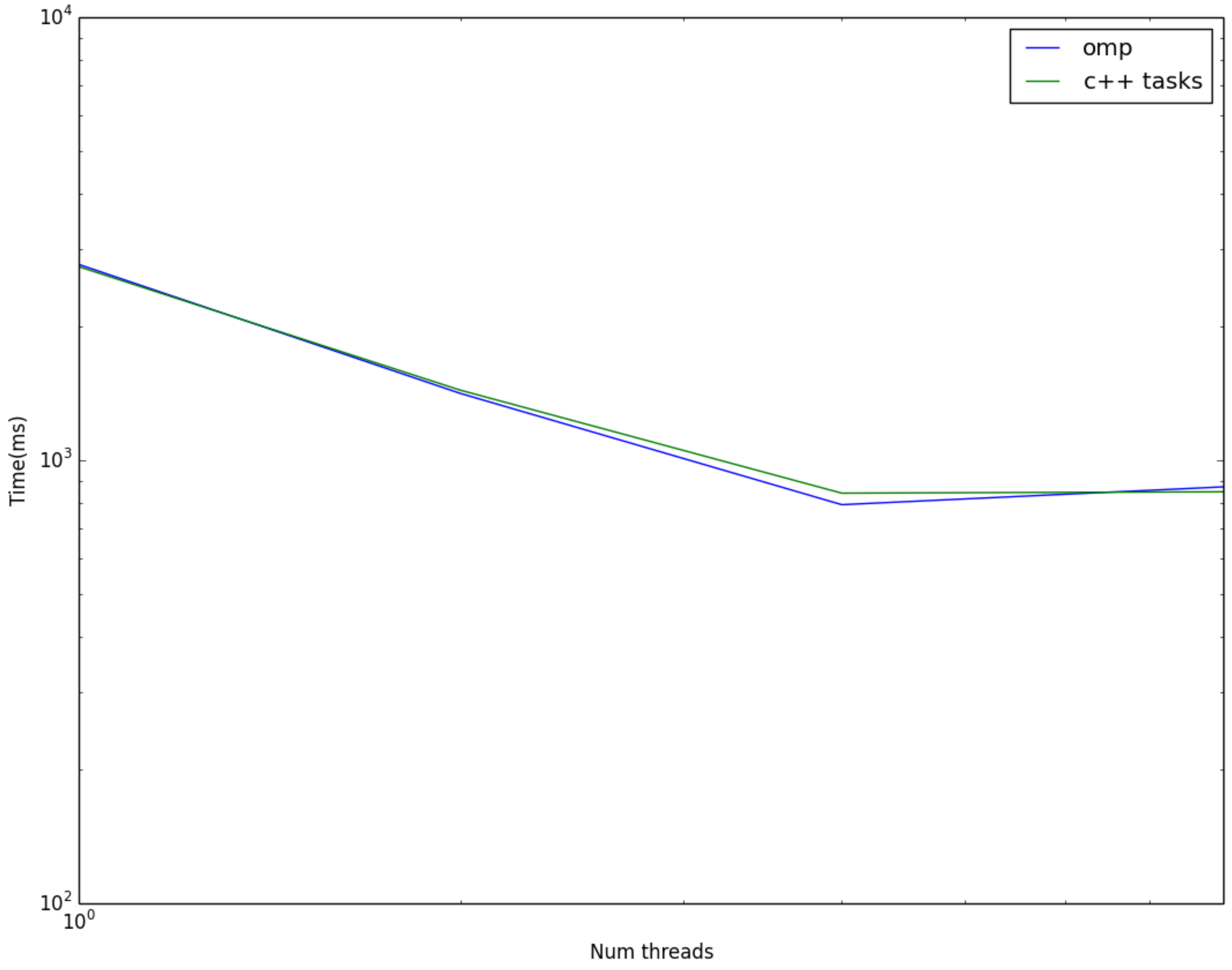
Add reduction over ”sum” with “+” *reduction-identifier*!

```
double two_norm(const Vector& x) {
  double sum = 0.0;
  #pragma omp parallel for reduction(+:sum)
  for (size_t i = 0; i < x.num_rows(); ++i) {
    sum += x(i) * x(i);
  }
  return std::sqrt(sum);
}
```

Finally

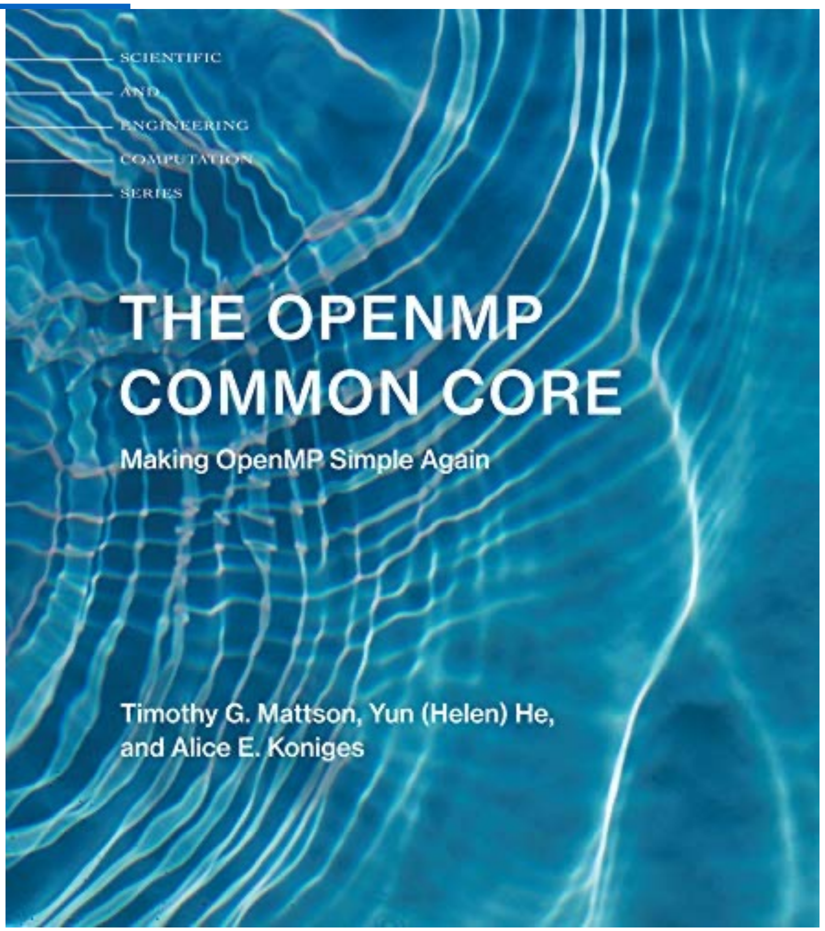
```
double two_norm(const Vector& x) {
  double sum = 0.0;
  #pragma omp parallel for reduction(+:sum)
  for (size_t i = 0; i < x.num_rows(); ++i) {
    sum += x(i) * x(i);
  }
  return std::sqrt(sum);
}
```

Performance



OpenMP Resources

- **OpenMP API Specification**
  - https://www.openmp.org/spec-html/5.0/openmp.html
- Book: The OpenMP Common Core
  - Tim Mattson, Helen He, Alice Koniges
- A "Hands-on" Introduction to OpenMP
  - A tutorial by Tim Mattson
  - https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES
THE OPENMP COMMON CORE
Making OpenMP Simple Again
Timothy G. Mattson, Yun (Helen) He, and Alice E. Koniges

Thank you!

Creative Commons BY-NC-SA 4.0 License