AMATH 483/583 High Performance Scientific Computing

Lecture 12: Tasks, async(), C++ Concurrency

Xu Tony Liu, PhD

Paul G. Allen School of Computer Science & Engineering

University of Washington

Seattle, WA

Overview

- In our last episode
  - Race condition
  - The critical-section problem
  - Atomic hardware instructions (Test and Set, Compare and Swap)
- Solutions of race condition
  - Mutex
  - Deadlock
  - Lock_guard
  - std::lock (avoid deadlock)
  - Asynchronous operation (std::async and std::future)
  - std::atomic (only working with integral type)

The Story So Far

In the beginning was the mainframe.
And the mainframe had a uniprocessor.
And tasks were scheduled in batches.

Then from the void there came Multics
And its pale imitator Unix
And there was pre-emptive multitasking

And tasks were given their own address spaces
And they were called processes.
And tasks were allowed to share memory
And they were called threads

The Story So Far

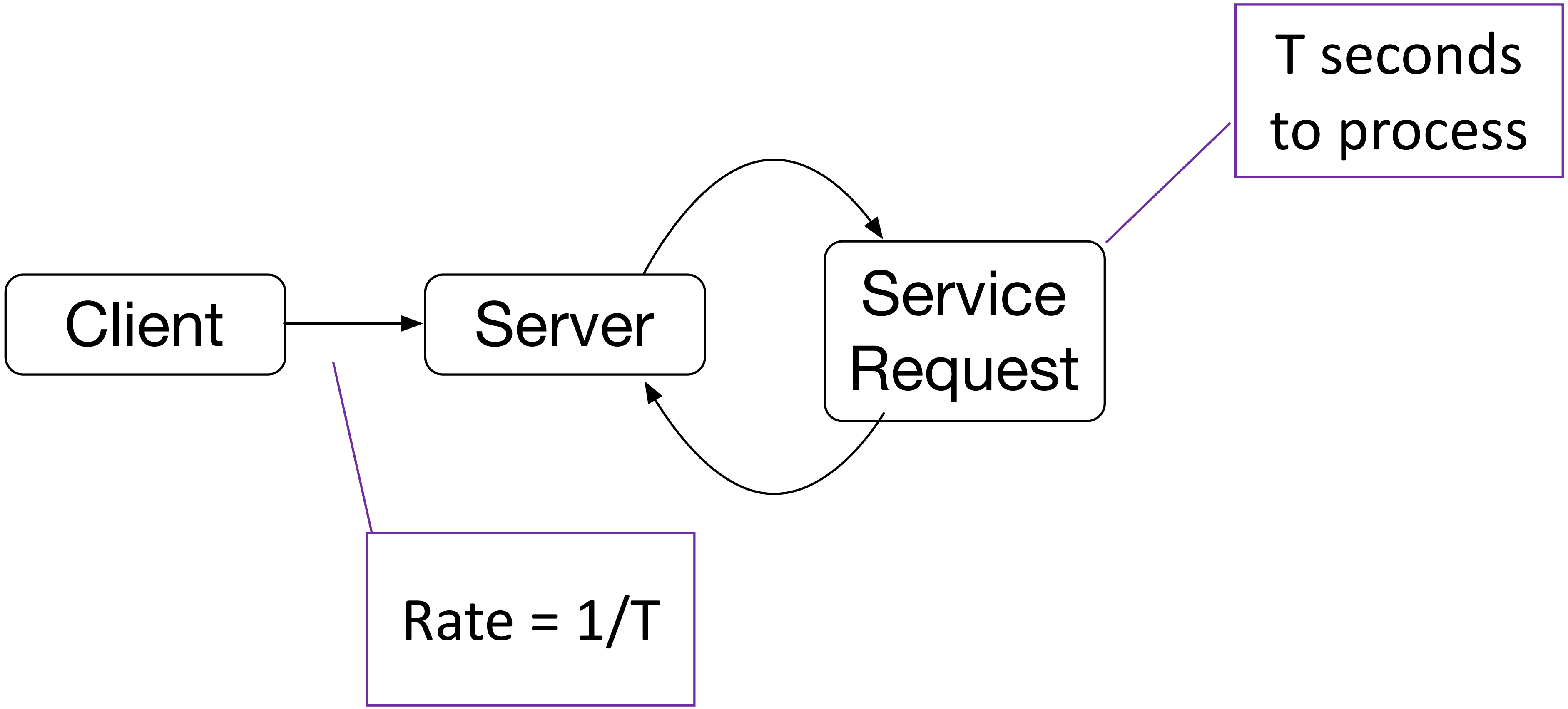
Then computers were given multiple CPUs
And multiple cores

And a multiplicity of concurrent tasks could run
At the same time
And there was parallelism

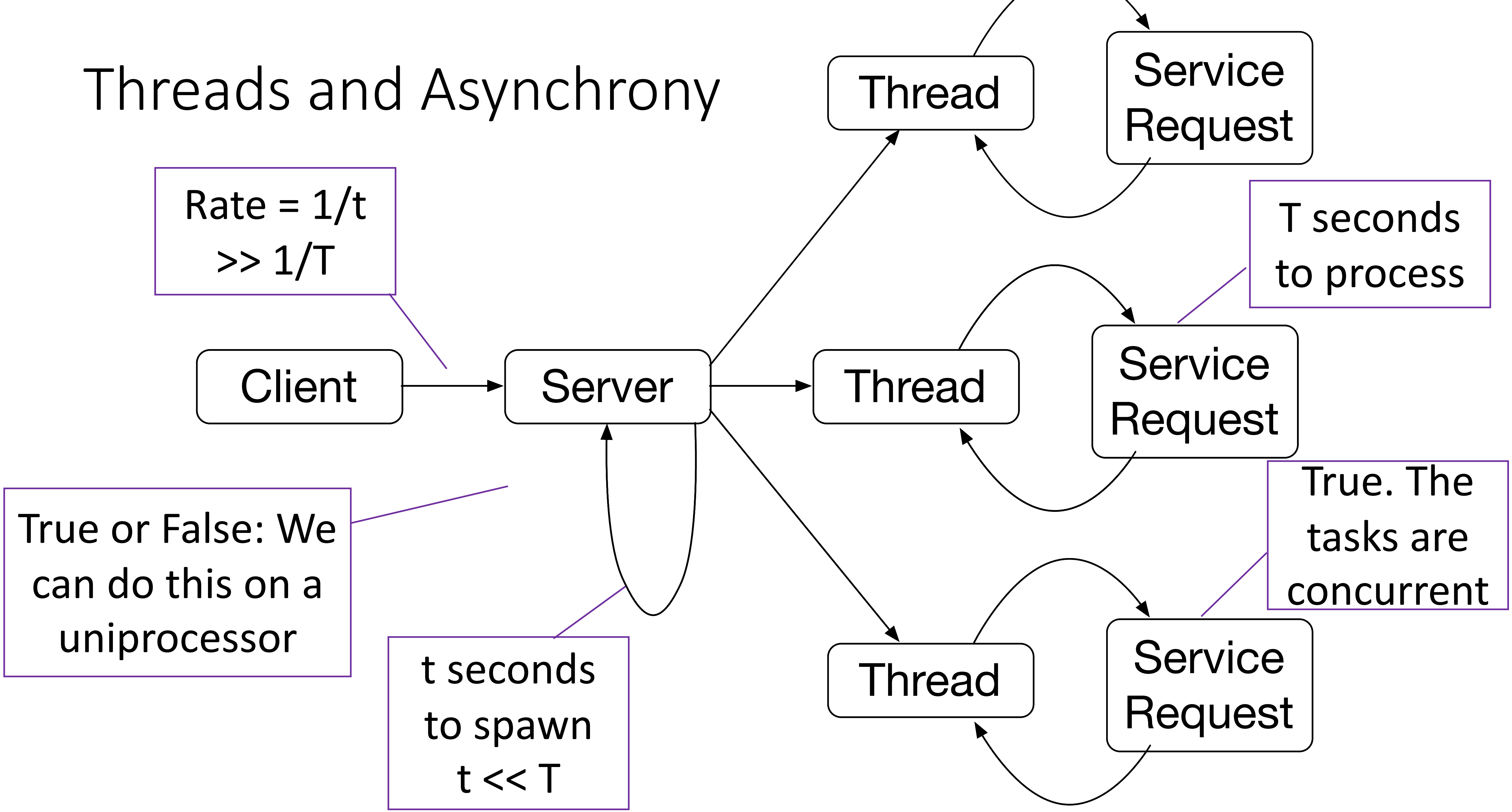
But in the shared memory there lurked race conditions
And other pernicious bugs

And lo, Dekker did give us his algorithm
To solve the critical section problem
And Dijkstra did give us semaphores and synchronization

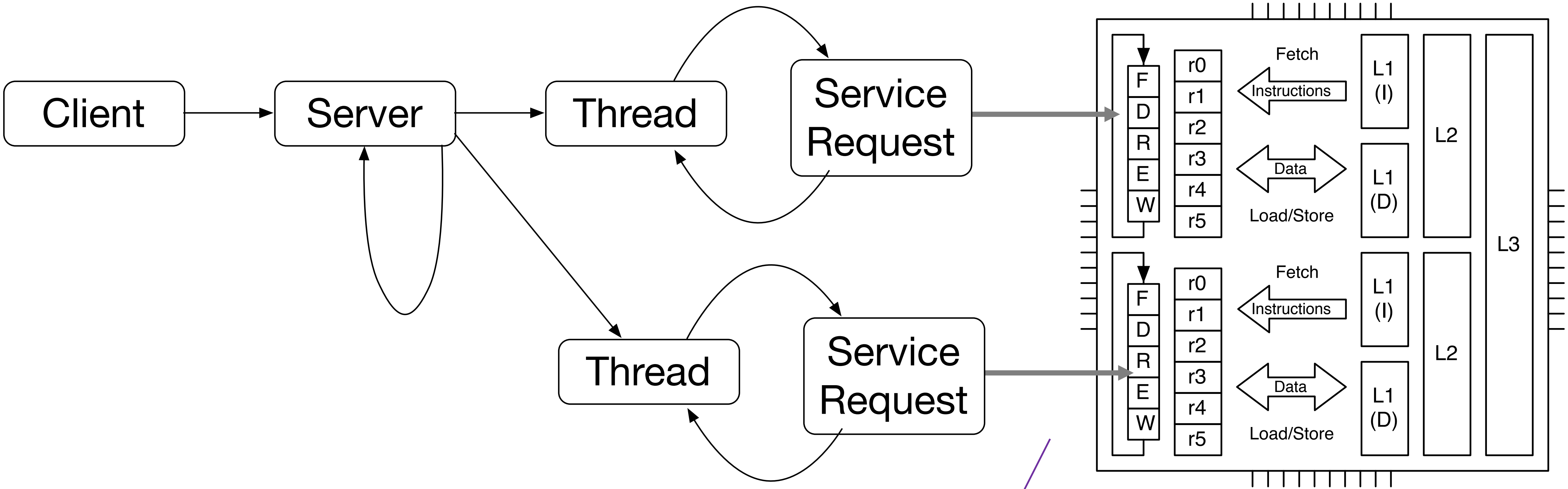
Threads and Asynchrony



Threads and Asynchrony



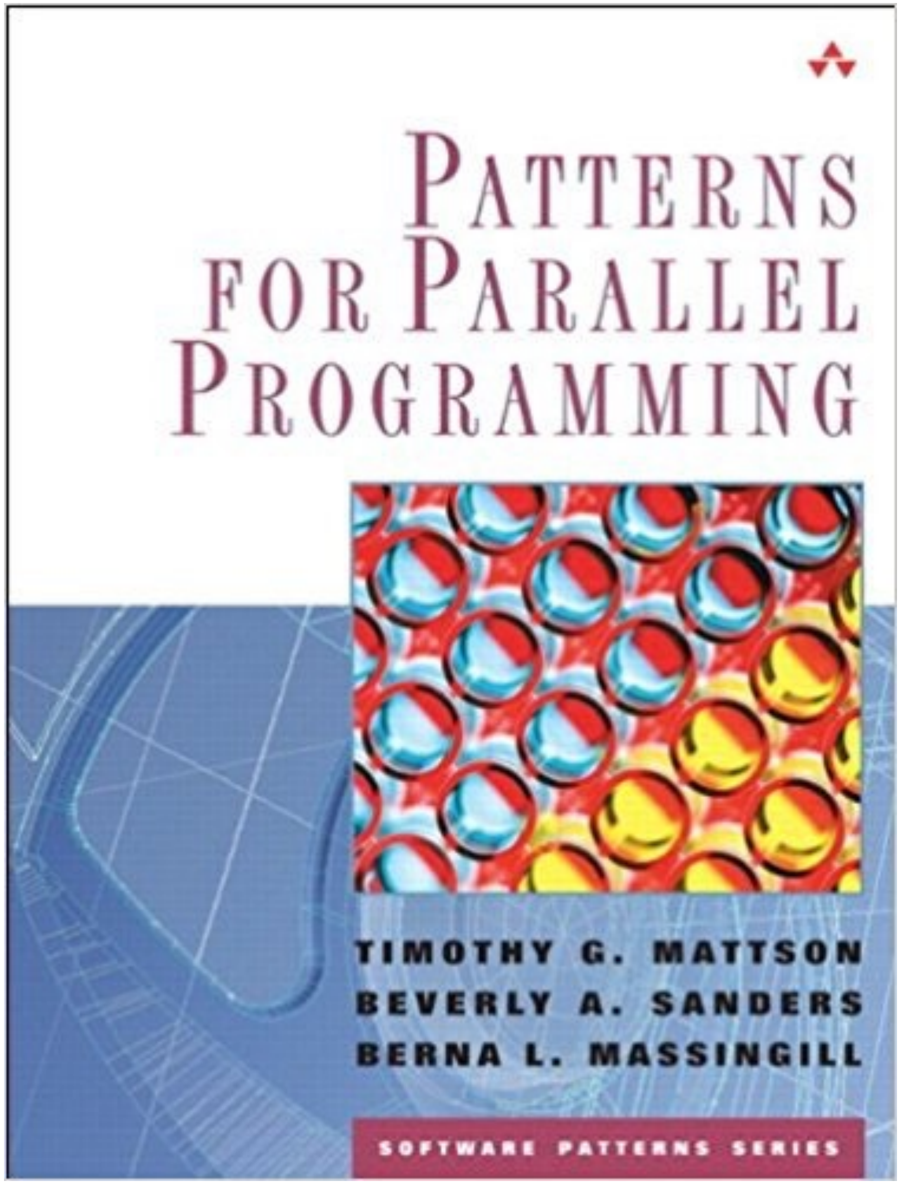
Multitasking on Multicore



On multiple cores, concurrent tasks can run in parallel

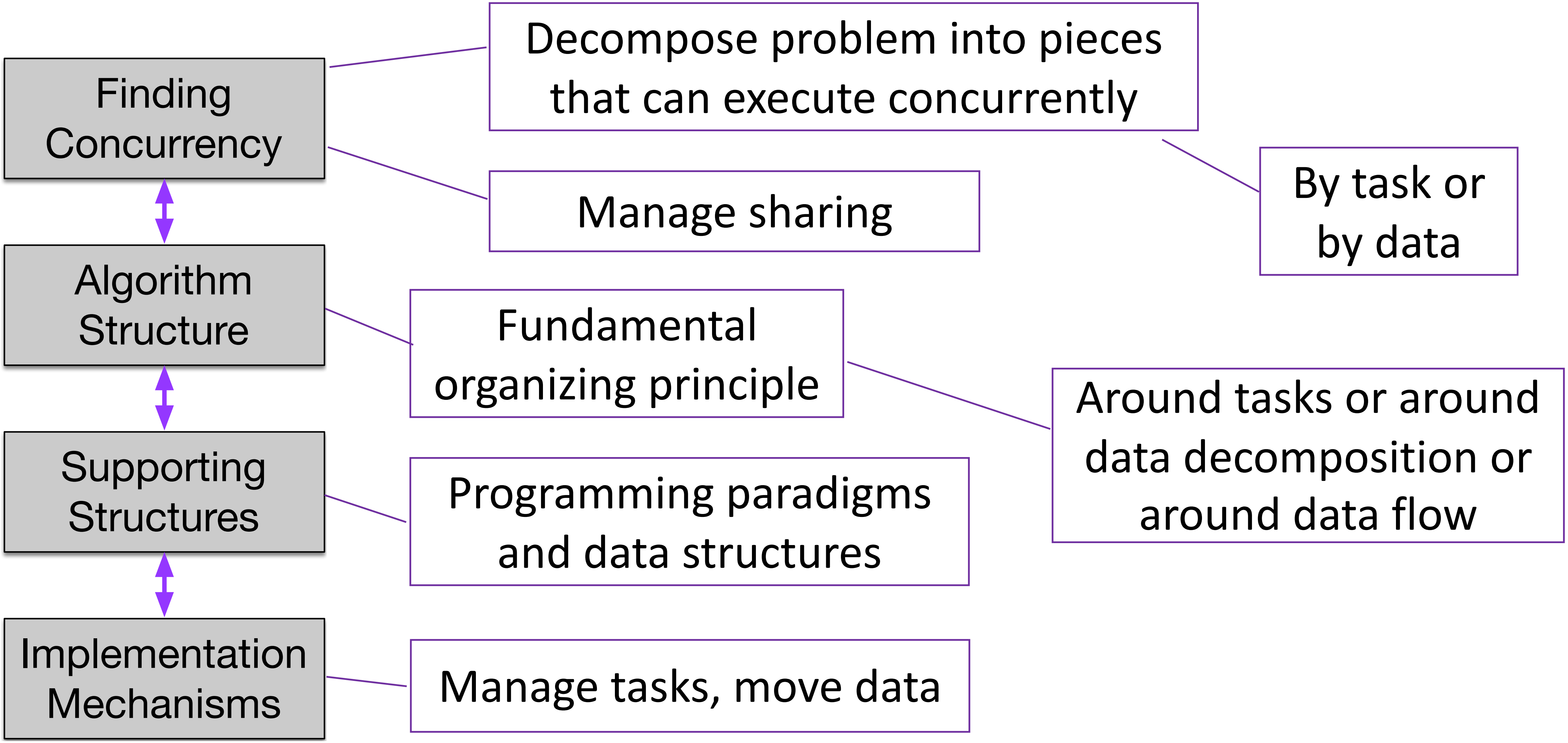
Parallelization Strategy

- How do we go from a problem we want to solve
- And maybe know how to solve sequentially
- To a parallel program
- That scales



Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*(First ed.). Addison-Wesley Professional.

Parallelization Strategy



Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*(First ed.). Addison-Wesley Professional.

Threads (Mechanism)

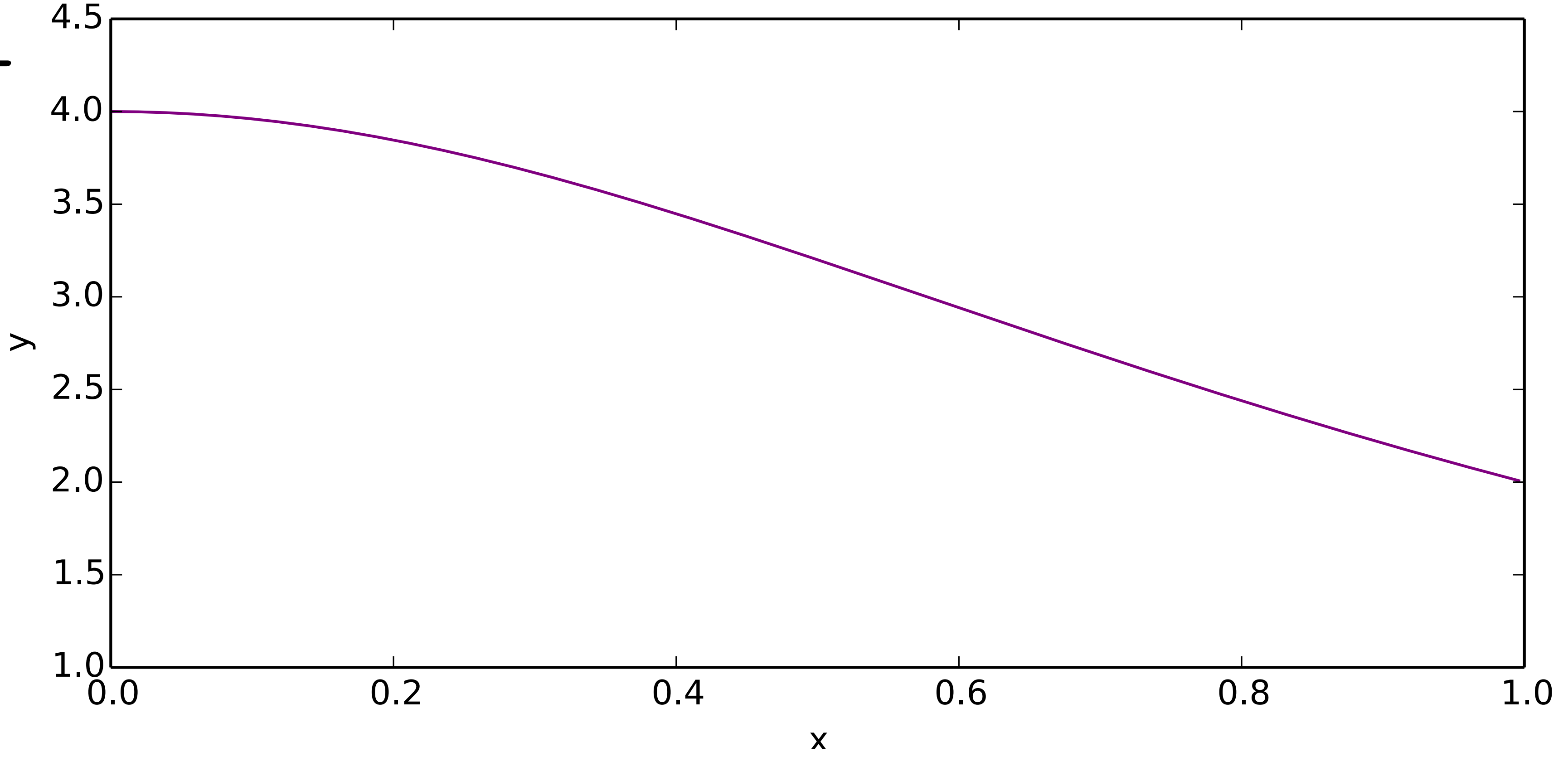
- How to launch a thread
- How to pass arguments to the thread’s task
- join() and detach()

Example

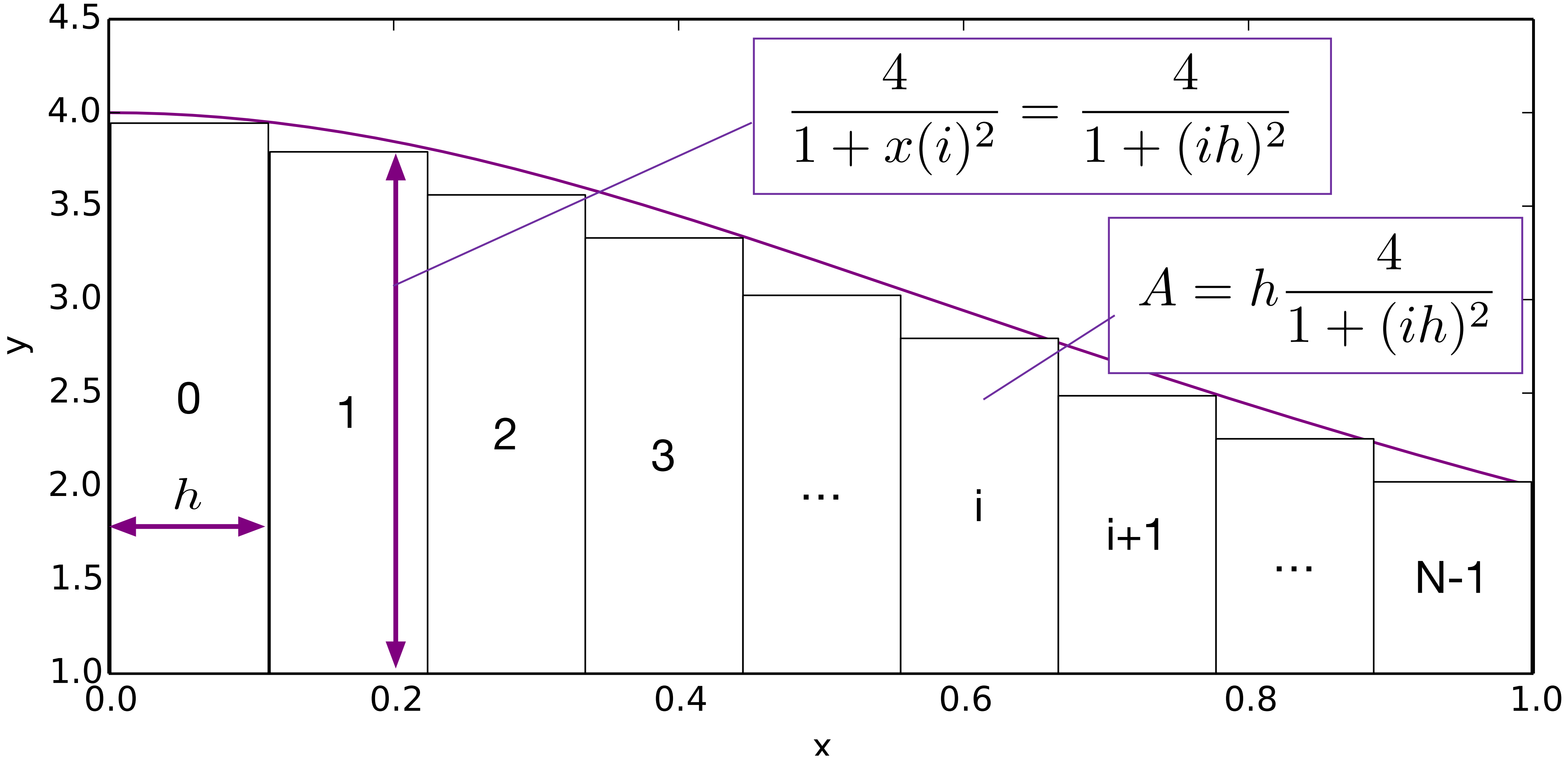
- Find the value of π

- Using formula

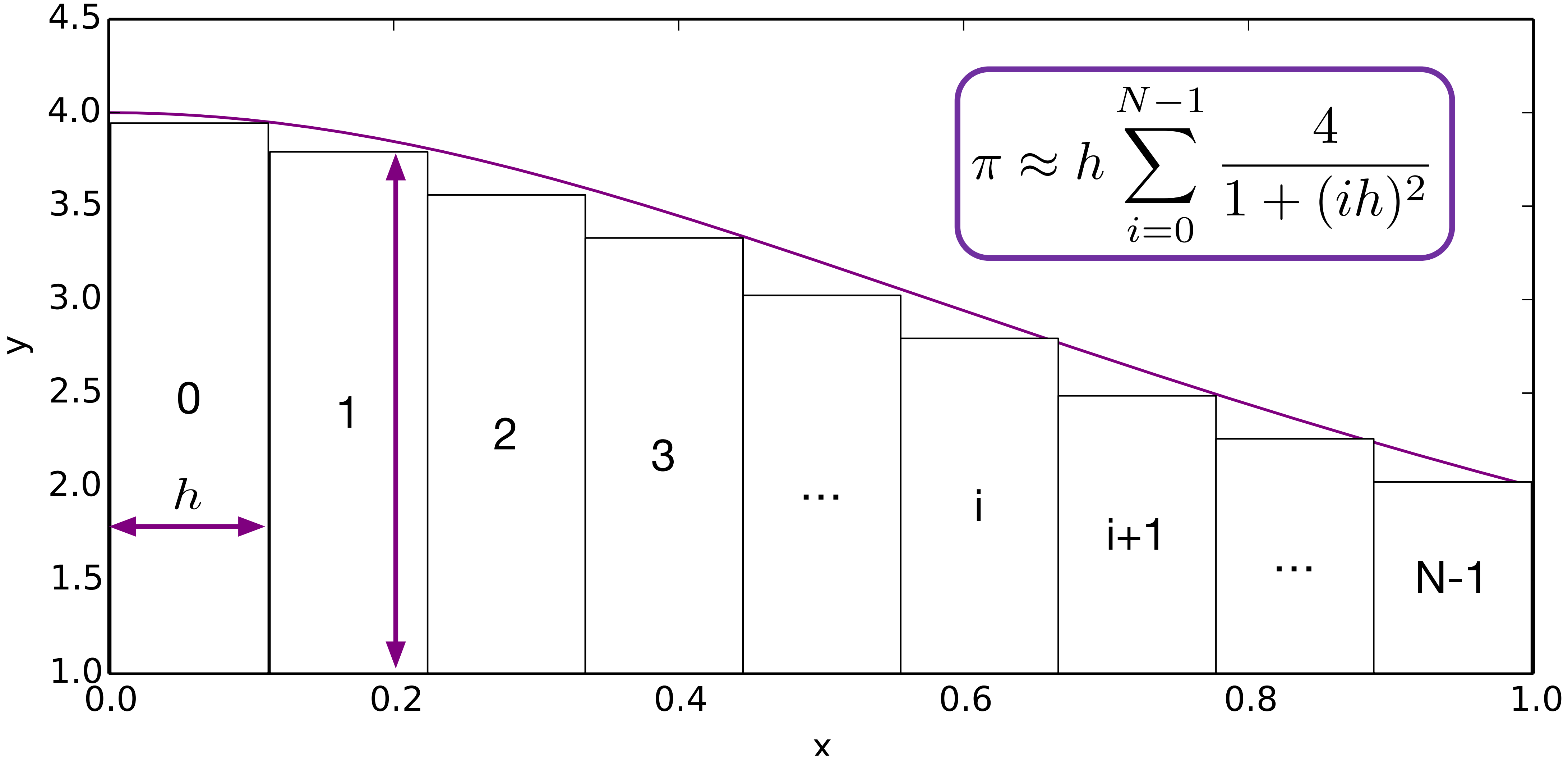
$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



Numerical Quadrature

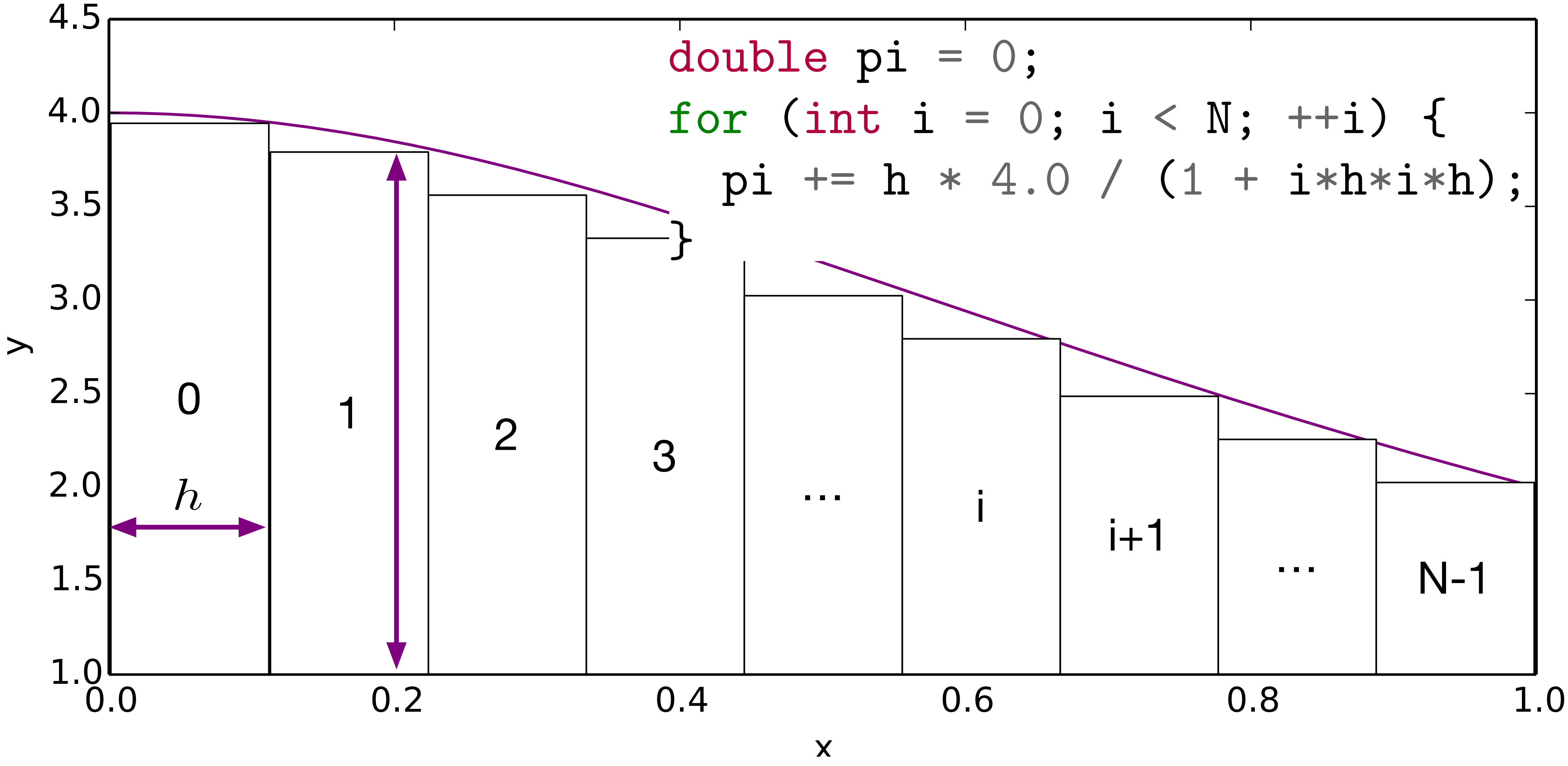


Numerical Quadrature



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

Numerical Quadrature (Sequential)

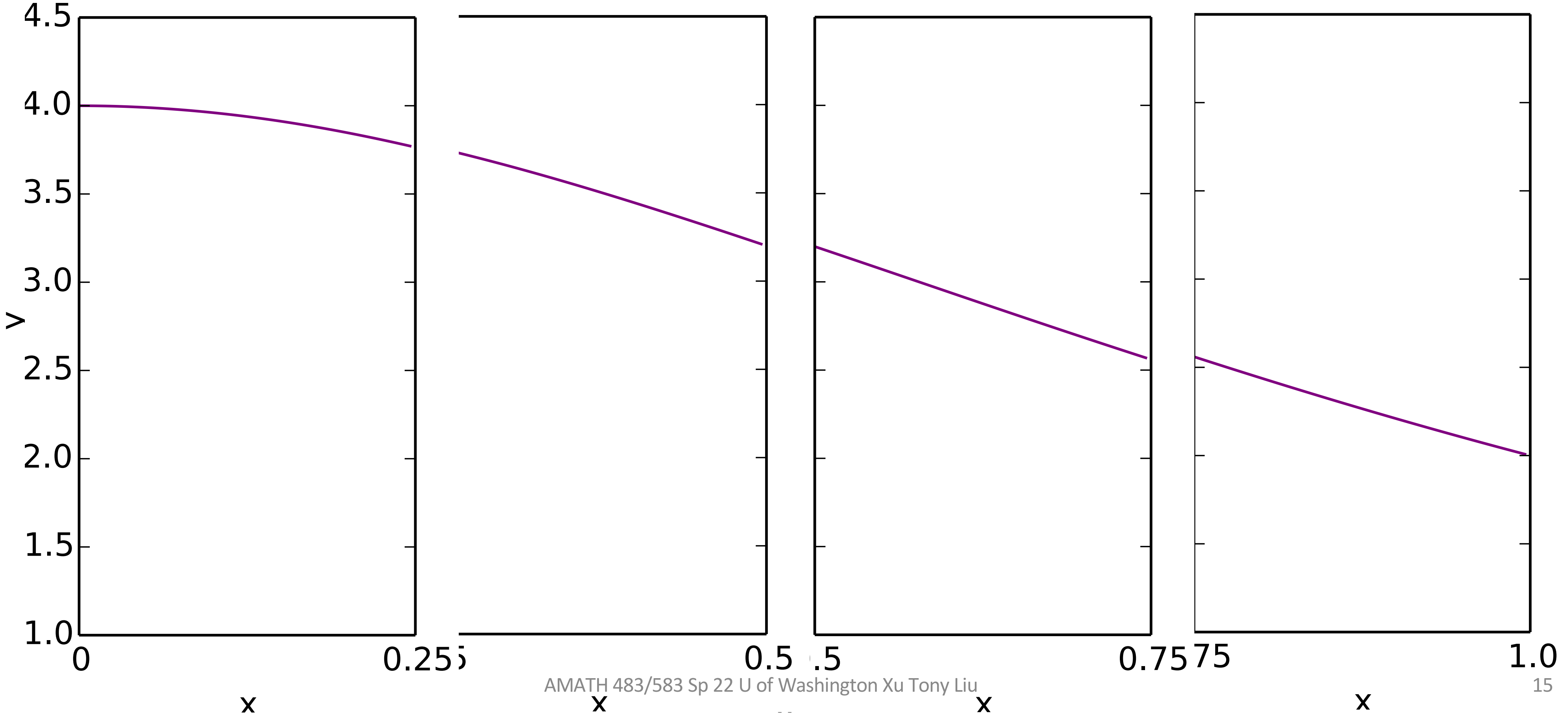



```
double pi = 0;
for (int i = 0; i < N; ++i) {
  pi += h * 4.0 / (1 + i*h*i*h);
}
```

Finding Concurrency

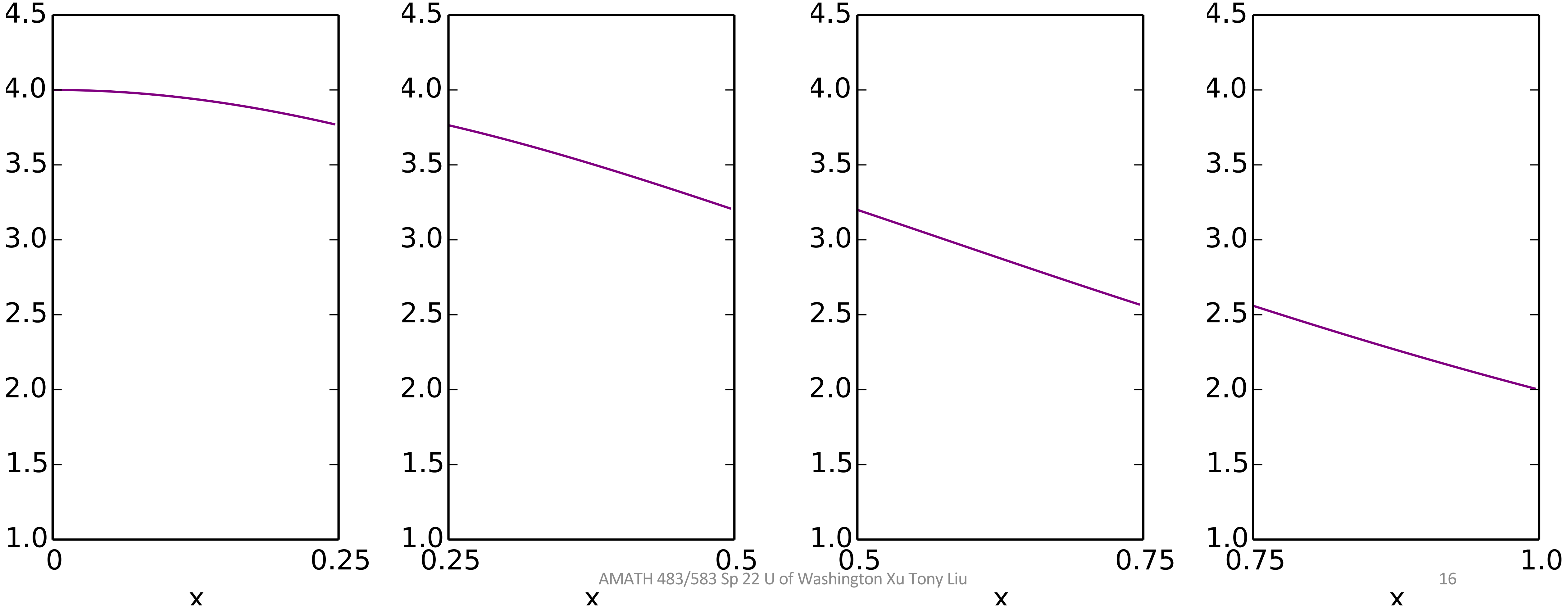
$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

$$\pi = \int_0^{0.25} \frac{4}{1+x^2}\,dx + \int_{0.25}^{0.5} \frac{4}{1+x^2}\,dx + \int_{0.5}^{0.75} \frac{4}{1+x^2}\,dx + \int_{0.75}^{1} \frac{4}{1+x^2}\,dx$$



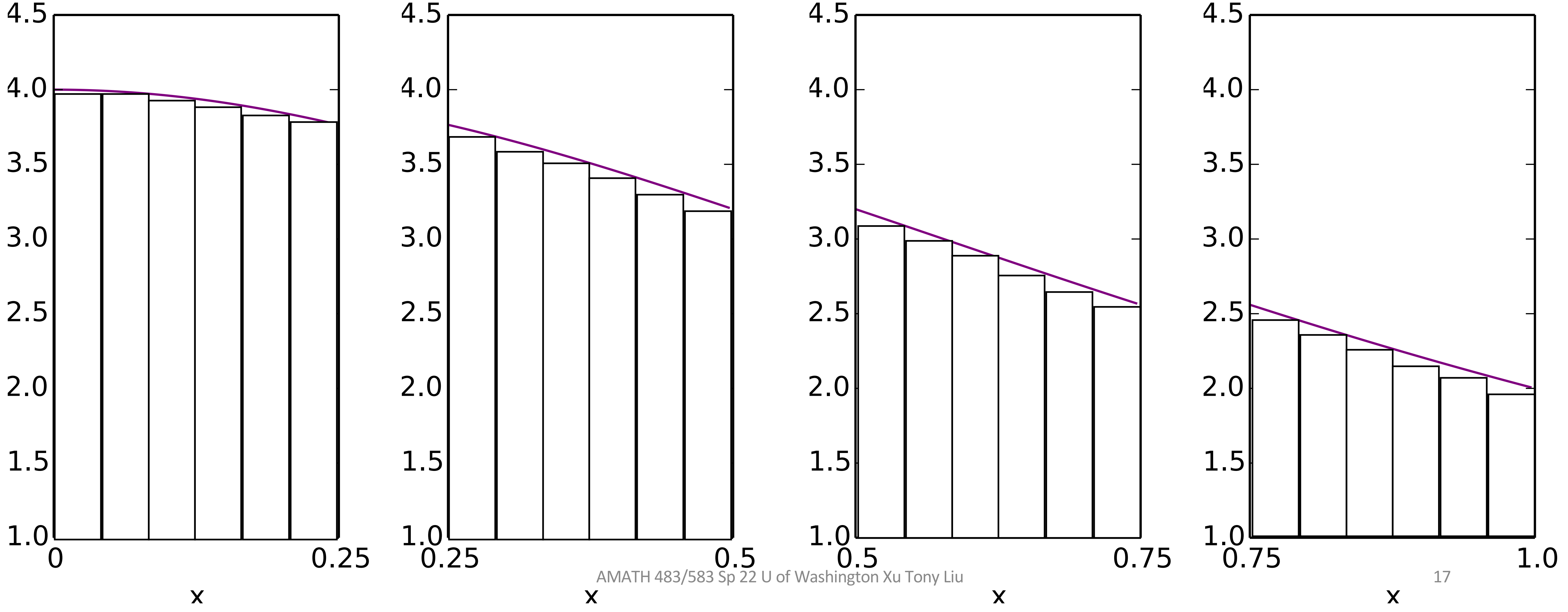
Finding Concurrency

$$\pi = \int_0^{0.25} \frac{4}{1+x^2}dx + \int_{0.25}^{0.5} \frac{4}{1+x^2}dx + \int_{0.5}^{0.75} \frac{4}{1+x^2}dx + \int_{0.75}^{1} \frac{4}{1+x^2}dx$$



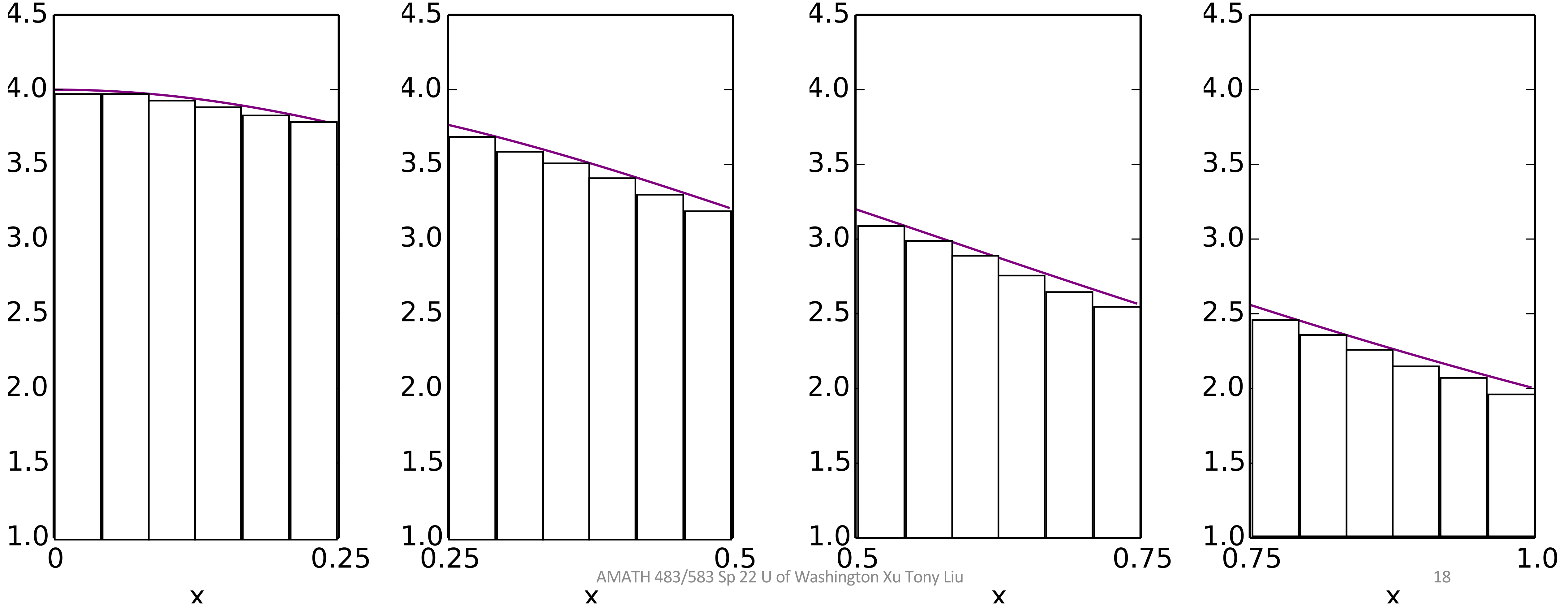
Finding Concurrency

$$\pi = \int_0^{0.25} \frac{4}{1+x^2}dx + \int_{0.25}^{0.5} \frac{4}{1+x^2}dx + \int_{0.5}^{0.75} \frac{4}{1+x^2}dx + \int_{0.75}^{1} \frac{4}{1+x^2}dx$$



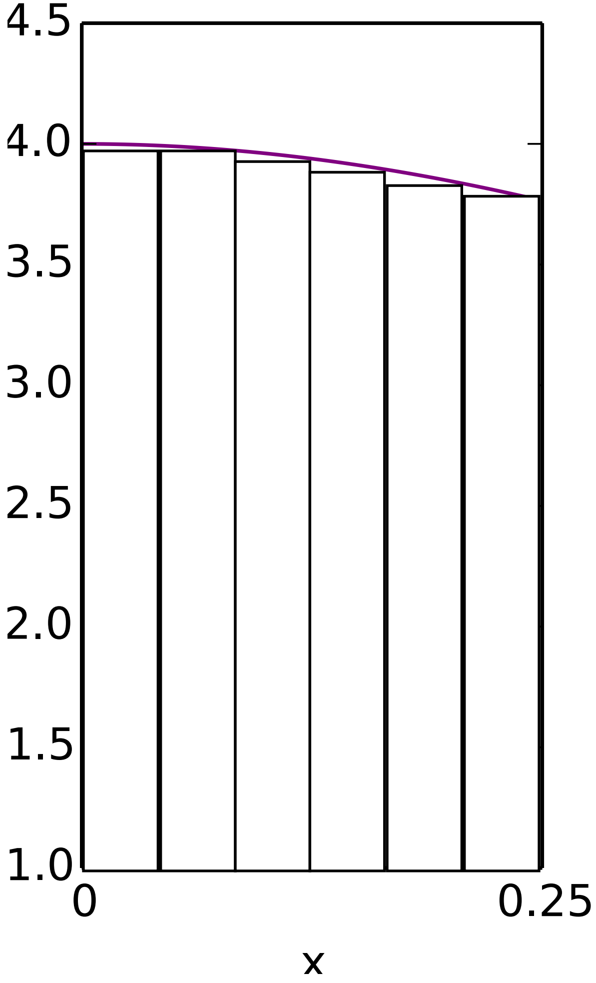
Finding Concurrency

$$\pi \approx h\sum_{i=0}^{N/4-1}\frac{4}{1+(ih)^2}+h\sum_{i=N/4}^{N/2-1}\frac{4}{1+(ih)^2}+h\sum_{i=N/2}^{3N/4-1}\frac{4}{1+(ih)^2}+h\sum_{i=3N/4}^{3N-1}\frac{4}{1+(ih)^2}$$

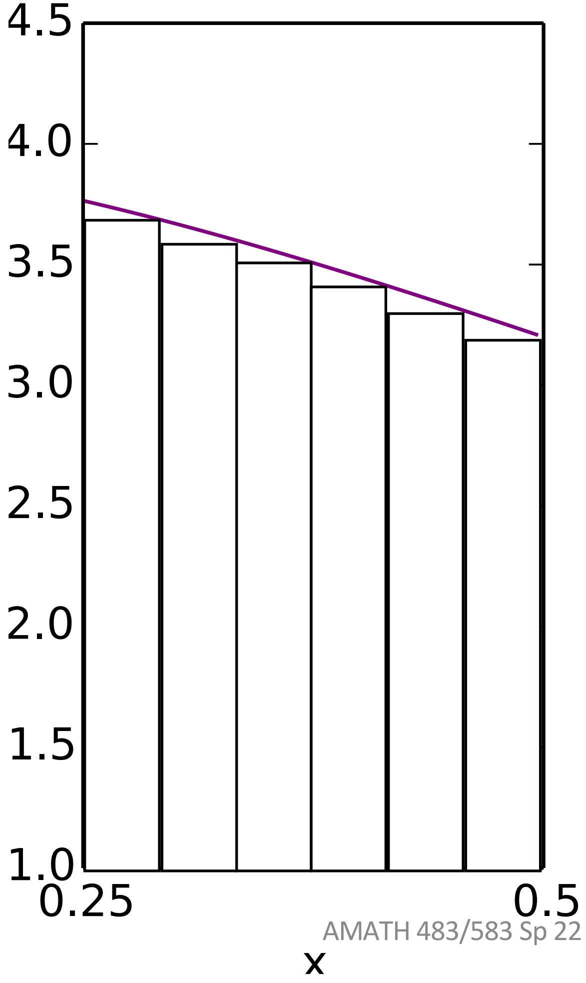


Finding Concurrency

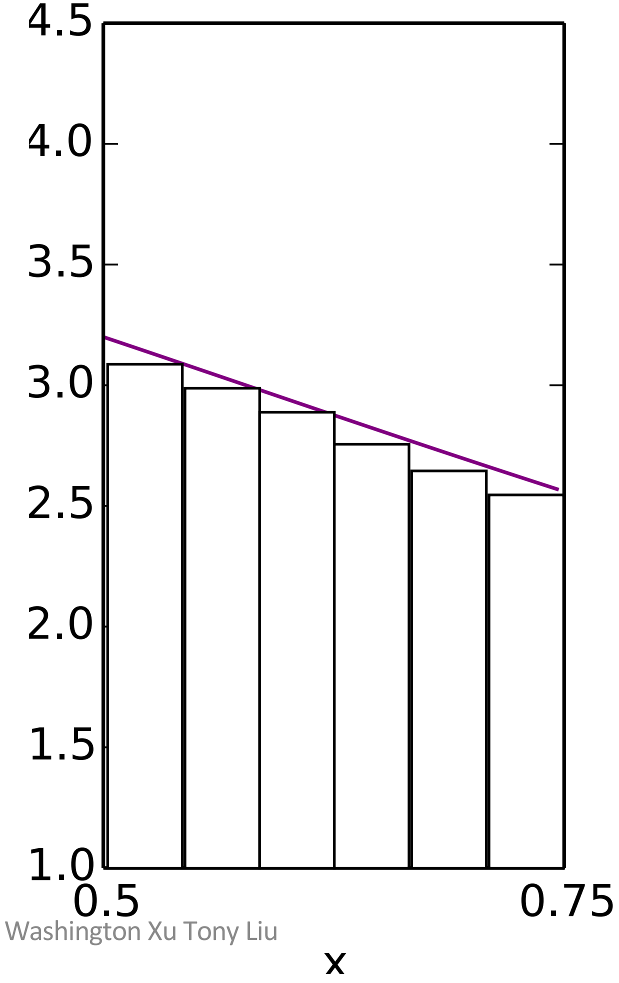
$$h\sum_{i=0}^{N/4-1}\frac{4}{1+(ih)^2}$$



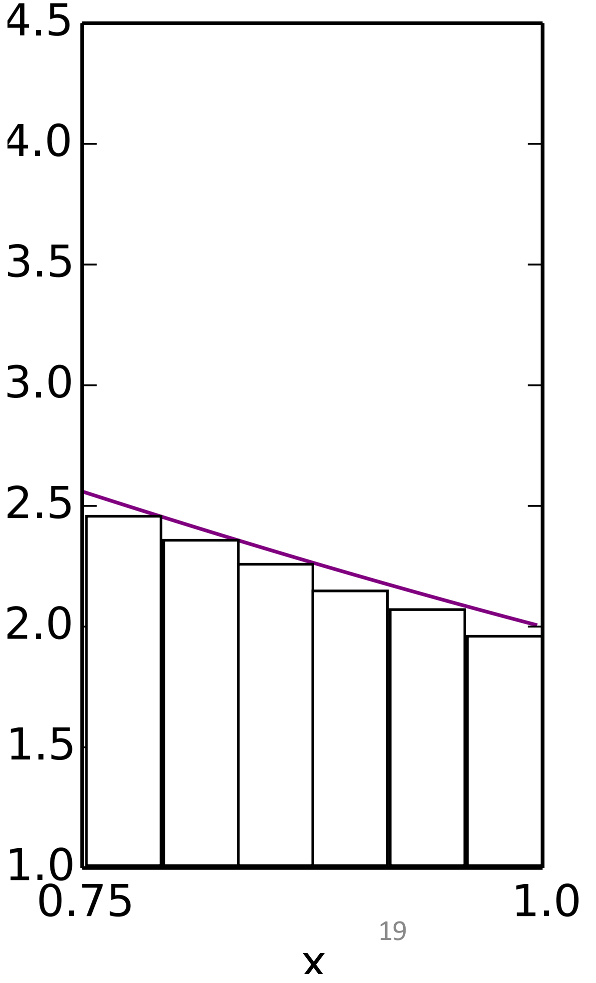
$$h\sum_{i=N/4}^{N/2-1}\frac{4}{1+(ih)^2}$$



$$h\sum_{i=N/2}^{3N/4-1}\frac{4}{1+(ih)^2}$$



$$h\sum_{i=3N/4}^{N-1}\frac{4}{1+(ih)^2}$$



Finding Concurrency

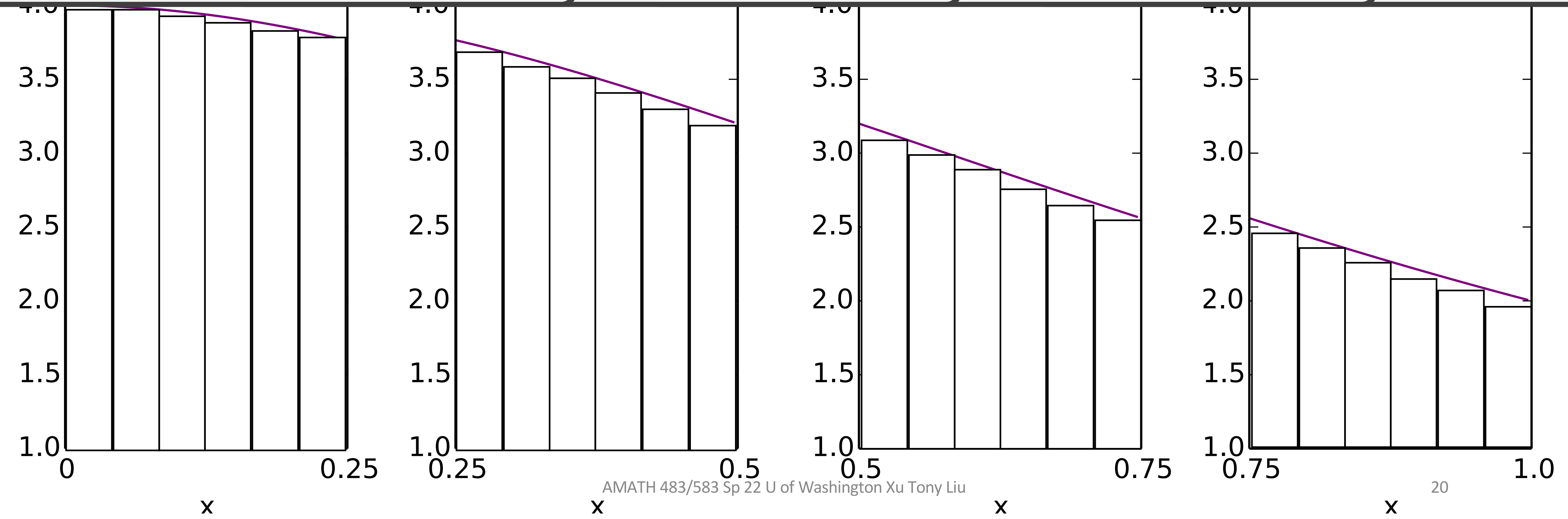
```
for (int i = begin; i < end; ++i) {
  pi += h * 4.0 / (1 + i*h*i*h);
}
```

```
int i = 0; i < N/4; ++i) {
+= h * 4.0 / (1 + i*h*i*h);
```

```
4; i < N/2; ++i) {
/ (1 + i*h*i*h);
```

```
/2; i < 3*N/4; ++i) {
0 / (1 + i*h*i*h);
```

```
N/4; i < N
/ (1 + i*h
```



Finding Concurrency

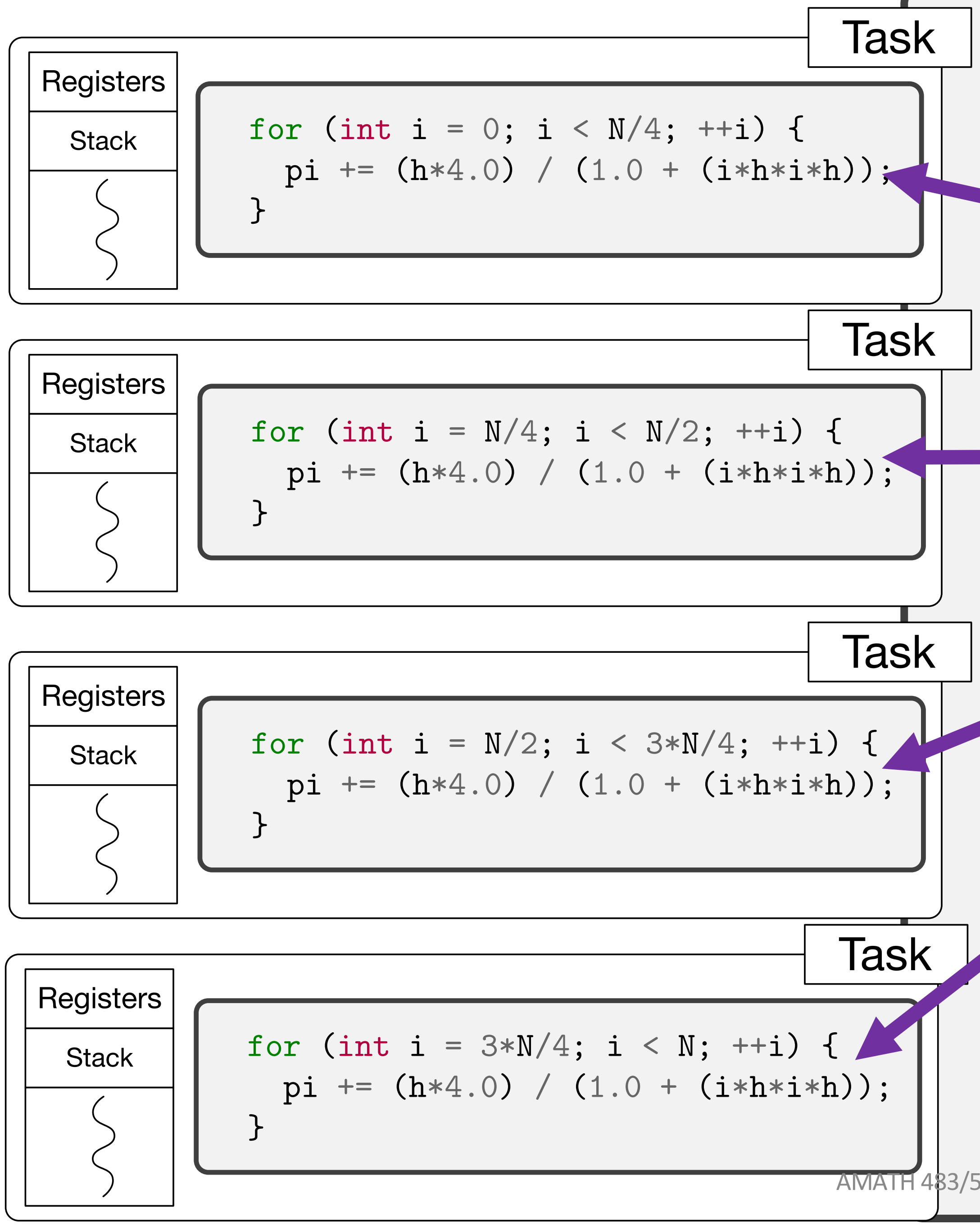
$$h \sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2}$$

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

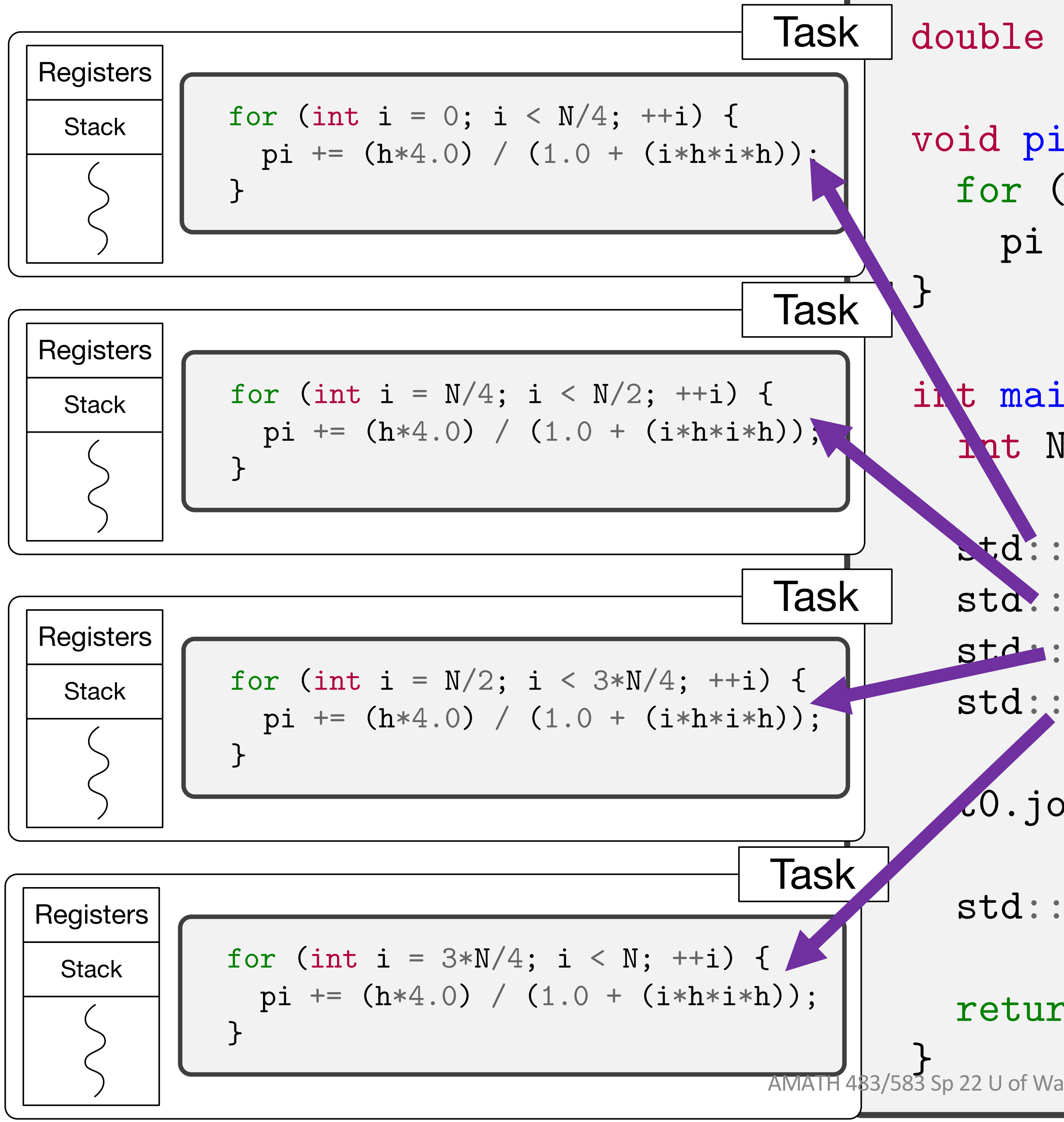
$$h \sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2}$$

$$h \sum_{i=3N/4}^{N-1} \frac{4}{1+(ih)^2}$$

```
int main() {
  double pi = 0.0;  int N = 1024*1024;

  for (int i = 0; i < N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```




```
int main() {
  double pi = 0.0;  int N = 1024*1024;

  for (int i = 0; i < N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```




```cpp
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/ (double)N;

  std::thread t0(pi_helper, 0,     N/4,   h);
  std::thread t1(pi_helper, N/4,   N/2,   h);
  std::thread t2(pi_helper, N/2,   3*N/4, h);
  std::thread t3(pi_helper, 3*N/4, N,     h);

  t0.join();  t1.join();  t2.join();  t3.join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

Threads

Function returning void

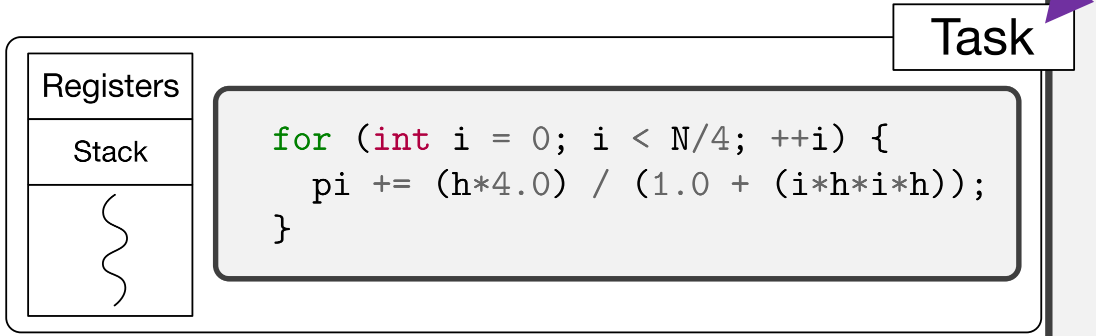
To run this function

Construct a thread

What if we want more or less than 4?

With these arguments

```cpp
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/ (double)N;

  std::thread t0(pi_helper, 0,       N/4,   h);
  std::thread t1(pi_helper, N/4,     N/2,   h);
  std::thread t2(pi_helper, N/2,     3*N/4, h);
  std::thread t3(pi_helper, 3*N/4,   N,     h);

  t0.join();  t1.join();  t2.join();  t3.join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```



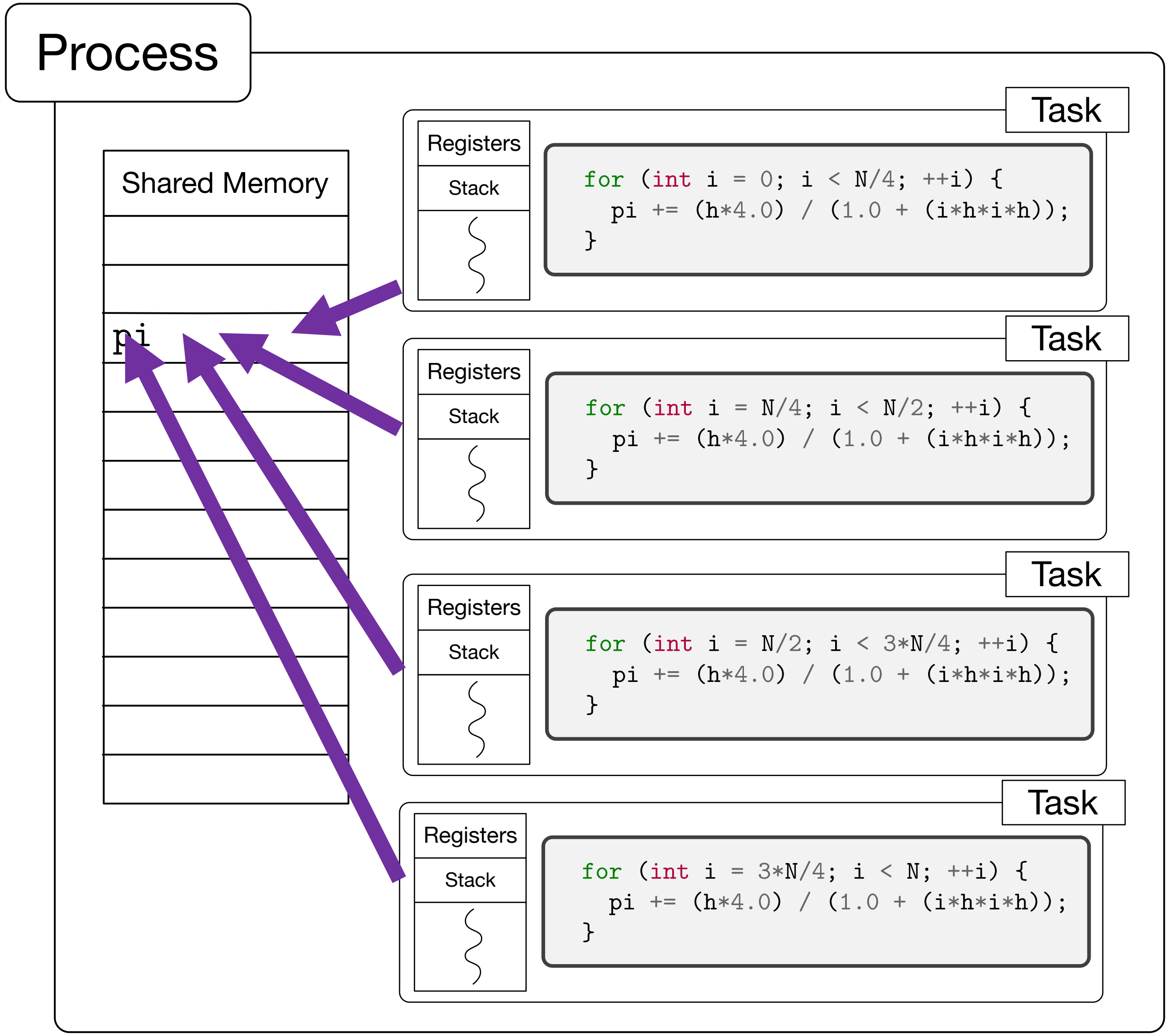
Threads

```
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/ (double)N;

  std::thread t0(pi_helper, 0,     N/4,   h);
  std::thread t1(pi_helper, N/4,   N/2,   h);
  std::thread t2(pi_helper, N/2,   3*N/4, h);
  std::thread t3(pi_helper, 3*N/4, N,     h);

  t0.join();  t1.join();  t2.join();  t3.join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

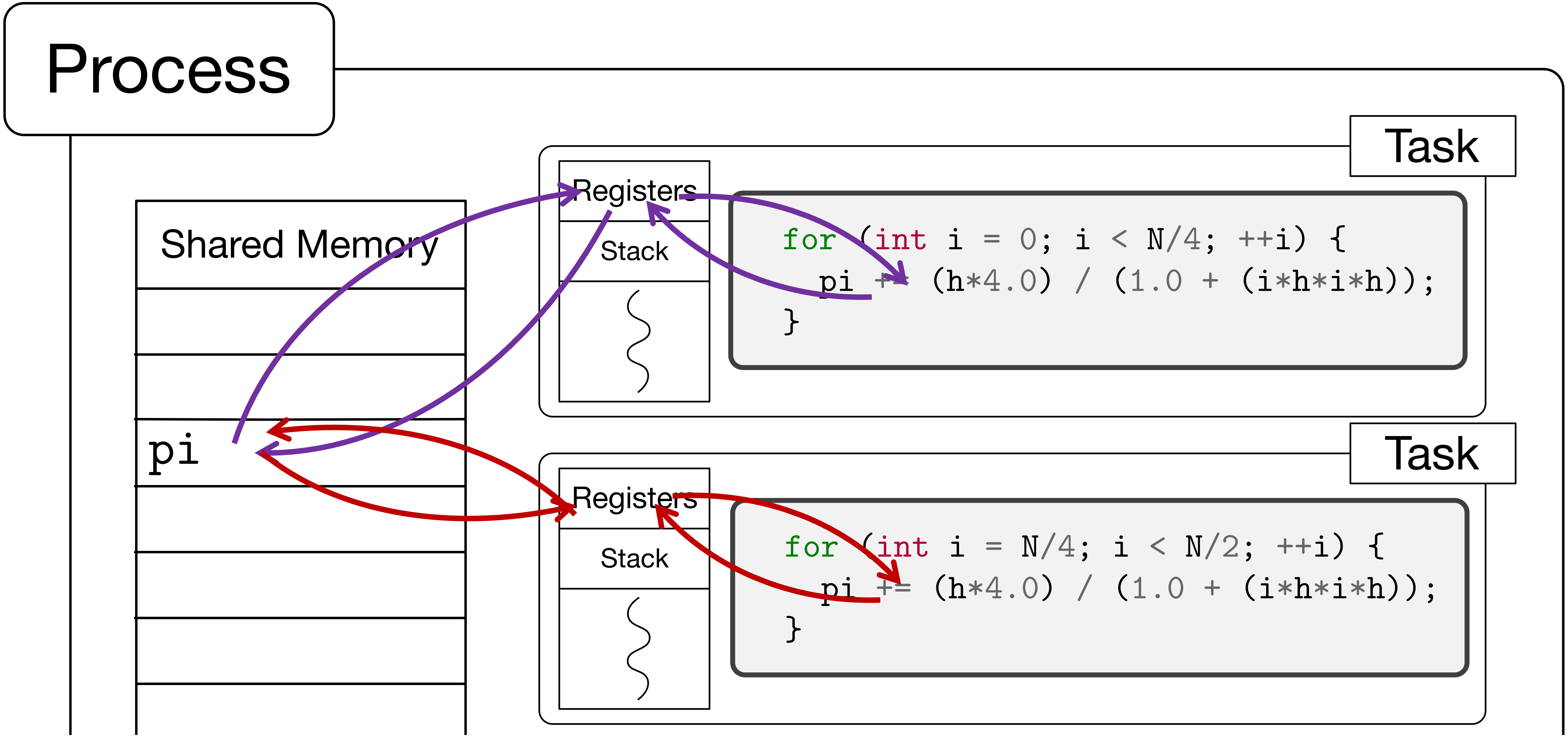
```
$ ./a.out
```

Threads

```
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/ (double)N;

  std::thread t0(pi_helper, 0,     N/4,   h);
  std::thread t1(pi_helper, N/4,   N/2,   h);
  std::thread t2(pi_helper, N/2,   3*N/4, h);
  std::thread t3(pi_helper, 3*N/4, N,     h);

  t0.join();  t1.join();  t2.join();  t3.join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```



Race Condition



Synchronization, Mutual Exclusion

- std::mutex
  - Lock and unlock
- Deadlock
- RAII -> lock_guard (*Resource Acquisition Is Initialization* or RAII)
  - When created, attempt to take ownership of the mutex it is given
  - When control leaves the scope, release the mutex
- std::lock
  - use a deadlock avoidance algorithm to avoid deadlock
  - Can lock multiple std::mutex objects

Mutex

```cpp
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i) {
    pi_mutex.lock();
    pi += (h*4.0) / (1.0 + (i*h*i*h));
    pi_mutex.unlock();
  }
}
```

Mutex

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  for (int i = begin; i < end; ++i) {
    pi_mutex.lock();
    pi += (h*4.0) / (1.0 + (i*h*i*h));
    pi_mutex.unlock();
  }
}
```

Locking and unlocking at every trip in inner loop

```
$ time ./a.out    # with race
```

Fast! But wrong!

Right! But slow!

Mutex

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  pi_mutex.lock();
  for (int i = begin; i < end; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
  }
  pi_mutex.unlock();
}
```

Locking and unlocking at every function call

```
$ time ./a.out    # with race
```

Fast! But wrong!

Fast! And right!

Mutex

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  pi_mutex.lock();
  for (int i = begin; i < end; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
  }
  pi_mutex.unlock();
}
```

Locking and unlocking at every function call

```
$ time ./a.out 1000000000
pi is ~ 6.24855709634561
3.680u 0.006s 0:03.68 100.0%
```

Right! But wrong!

Really big number

Wait. What?

Integers

Equivalent type	Width in bits by data model				
	C++ standard	LP32	ILP32	LLP64	LP64
`short` `unsigned short`	at least **16**	**16**	**16**	**16**	**16**
`int` `unsigned int`	at least **16**	**16**	**32**	**32**	**32**
`long` `unsigned long`	at least **32**	**32**	**32**	**32**	**64**
`long long` `unsigned long long`	at least **64**	**64**	**64**	**64**	**64**

Types

```
template <typename T>
void out_type_info() {
  std::cout << typeid(T).name()            << "\t";
  std::cout << sizeof(T)
  std::cout << 8*sizeo
  std::cout << std::nu
  std::cout << std::nu
}

int main() {
  std::cout << "Type\t
  out_type_info<bool>
  out_type_info<int>()
  out_type_info<unsign
  out_type_info<long>
  out_type_info<unsign
  out_type_info<long l
  out_type_info<unsigned
  out_type_info<float>();
  out_type_info<double>();

  return 0;
}
```

Type	Bytes	Bits	Min	Max
b	1	8	0	1
i	4	32	-2147483648	2147483647
j	4	32	0	4294967295
l	8	64	-9223372036854775808	9223372036854775807
m	8	64	0	18446744073709551615
x	8	64	-9223372036854775808	9223372036854775807
y	8	64	0	18446744073709551615
f	4	32	1.17549e-38	3.40282e+38
d	8	64	2.22507e-308	1.79769e+308

2 billion - big?

Mutex

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(unsigned long begin, unsigned long end, double h) {
  pi_mutex.lock();
  for (unsigned long i = begin; i < end; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
  }
  pi_mutex.unlock();
}
```

```
$ time ./a.out 1000000000
pi is ~ 3.14159265458933
2.036u 0.003s 0:02.04 99.5%
```

Locking and unlocking at every function call

unsigned long

Right!

Really big number

Mutex

Locking and unlocking at every function call

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(unsigned long begin, unsigned long end, double h) {
  pi_mutex.lock();
  for (unsigned long i = begin; i < end; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
  }
  pi_mutex.unlock();
}
```

```
$ time ./a.out 1000000000 # sequential
pi is ~ 3.14159265458978
2.013u 0.003s 0:02.01 100.0%
```

Right! And fast! But not scaling!

Why not?

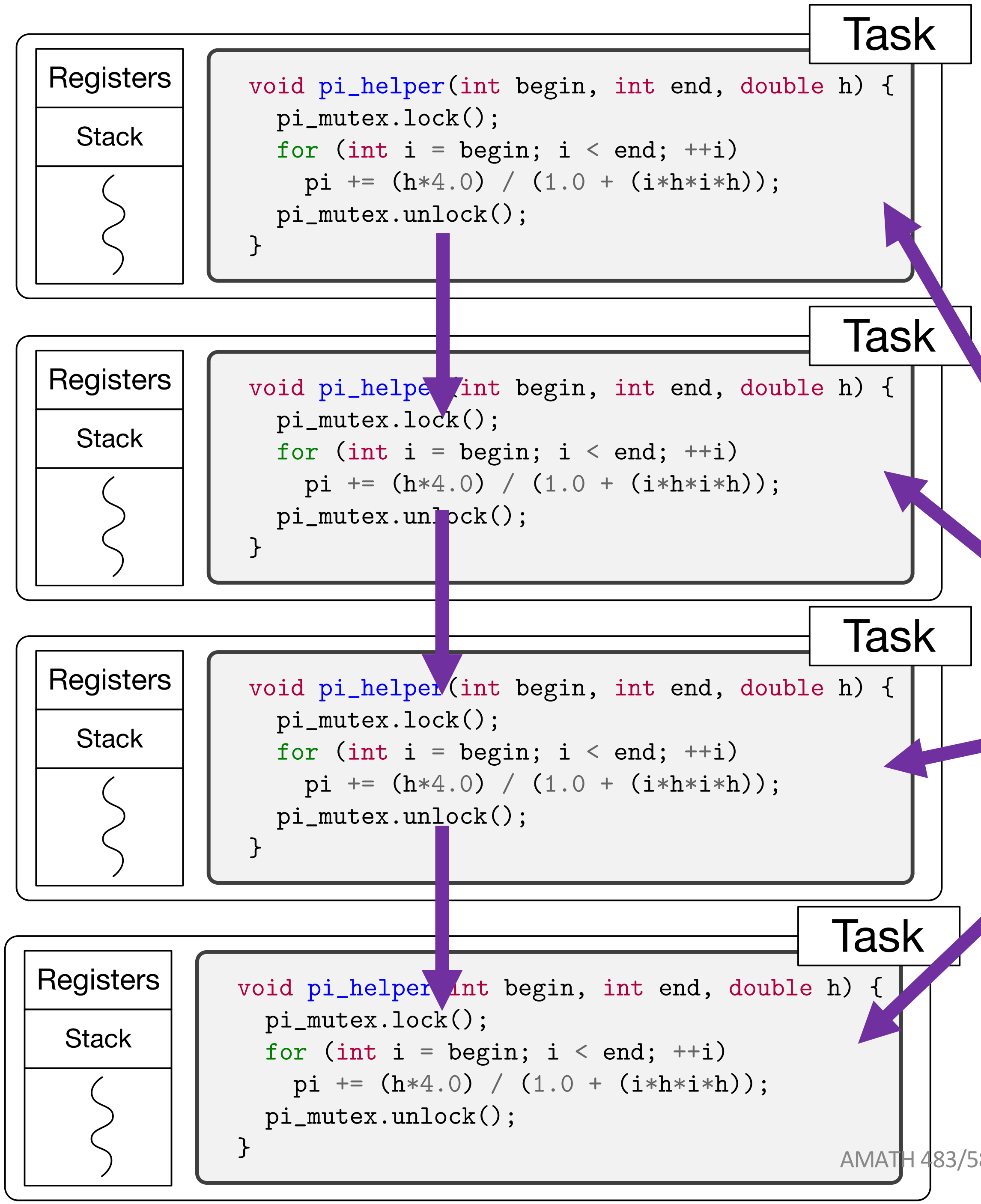
Mutex

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(unsigned long begin, unsigned long end, double h) {
  pi_mutex.lock();
  for (unsigned long i = begin; i < end; ++i) {
    pi += (h*4.0) / (1.0 + (i*h*i*h));
  }
  pi_mutex.unlock();
}
```

Locking and unlocking at every function call

Can multiple threads run this in parallel? (or even concurrently?)

```
$ time ./a.out 1000000000 # sequential
pi is ~ 3.14159265458978
2.013u 0.003s 0:02.01 100.0%
```




```cpp
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  pi_mutex.lock();
  for (int i = begin; i < end; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));
  pi_mutex.unlock();
}

int main(int argc, char* argv[]) {
  int N = 1024 * 1024; double h = 1.0/(double) N;

  std::thread t0(pi_helper, 0,     N/4,   h);
  std::thread t1(pi_helper, N/4,   N/2,   h);
  std::thread t2(pi_helper, N/2,   3*N/4, h);
  std::thread t3(pi_helper, 3*N/4, N,     h);

  t0.join();  t1.join();  t2.join();  t3.join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

Back Where We Started

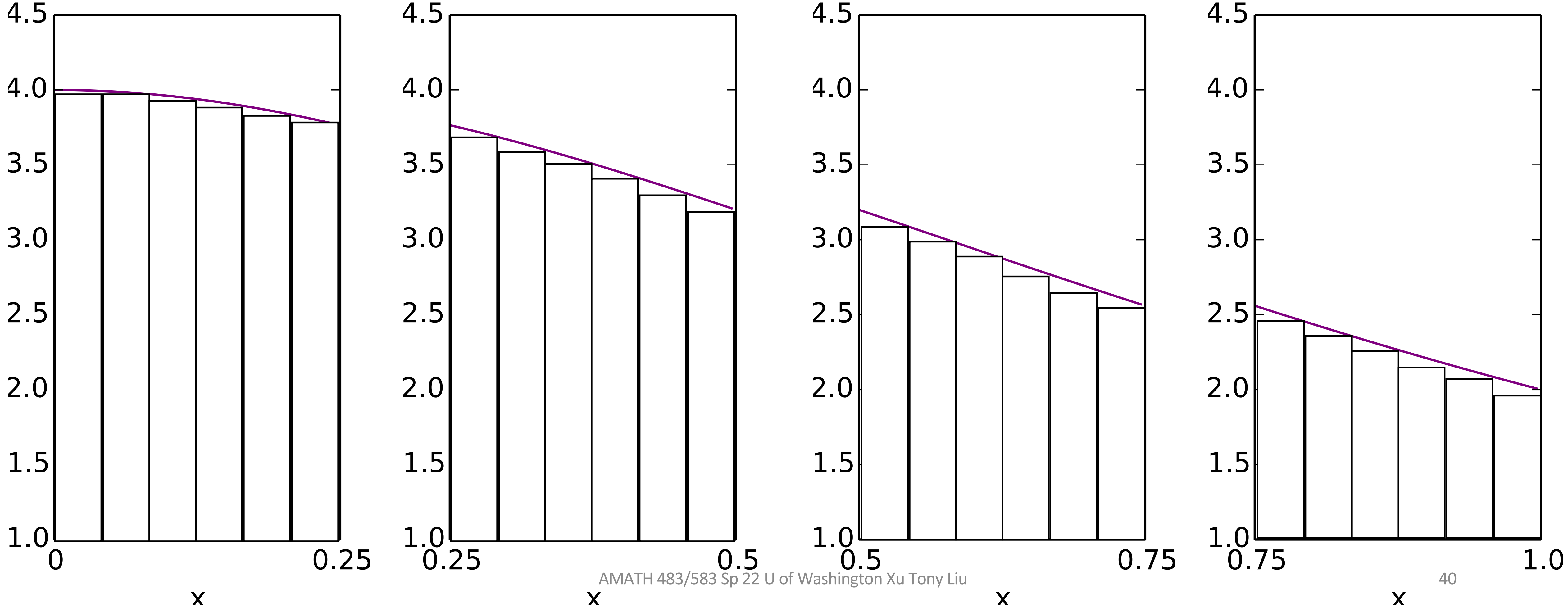
- What happened?
- We found concurrency (partitioned the integration)
- We had a race b/c shared pi
- Protected each update
- Too slow
- Protected each helper
- No longer concurrent

```
int main() {
  double pi = 0.0;   int N = 1024*1024;

  for (int i = 0; i < N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```

Finding Concurrency

$$\pi = \pi_0 + \pi_1 + \pi_2 + \pi_3$$

$$\pi_0 = h \sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2} \quad \pi_1 = h \sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2} \quad \pi_2 = h \sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2} \quad \pi_3 = h \sum_{i=3N/4}^{N-1} \frac{4}{1+(ih)^2}$$



Finding Concurrency

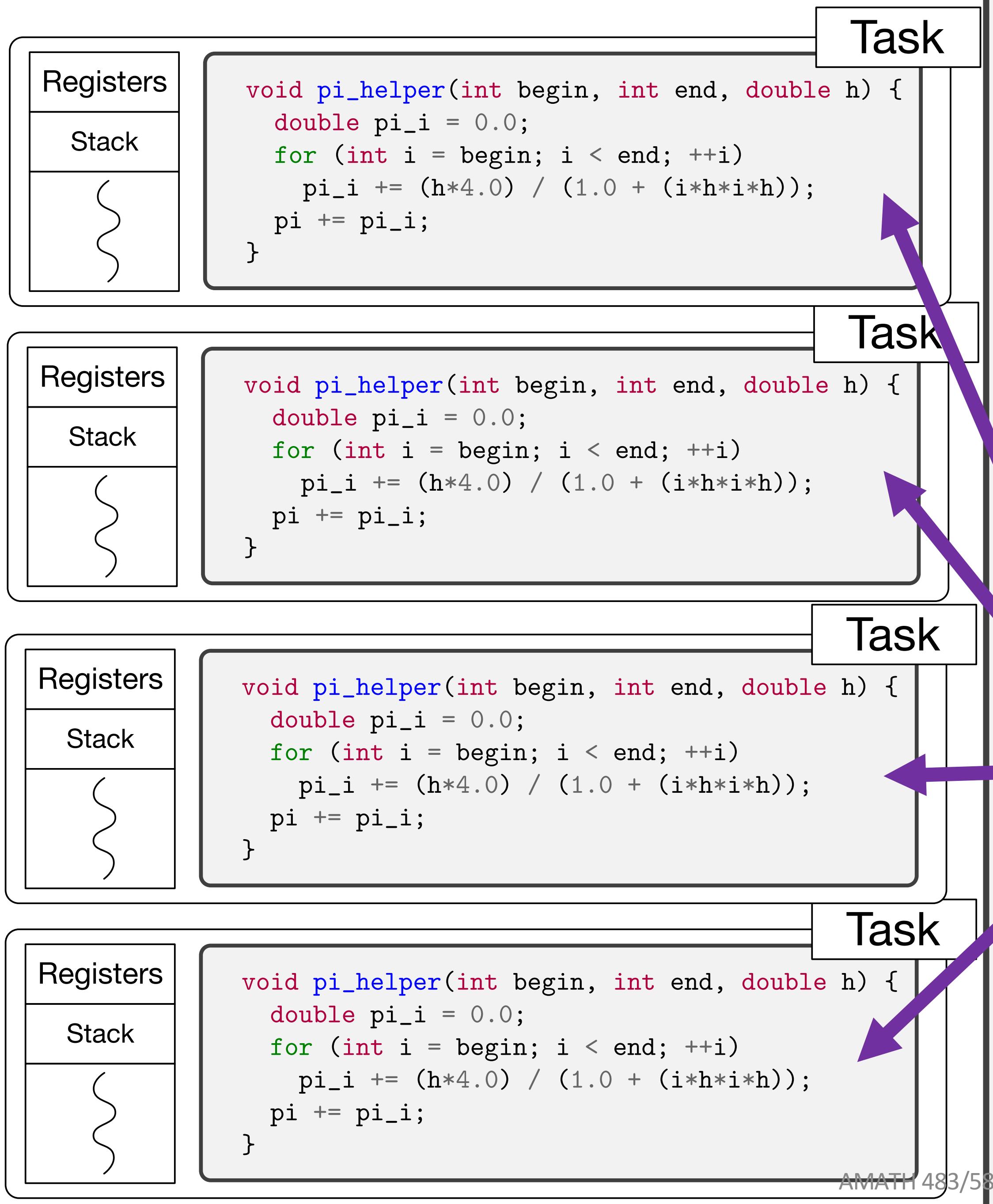
$$\pi_0 = h \sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2}$$

$$\pi_1 = h \sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2}$$

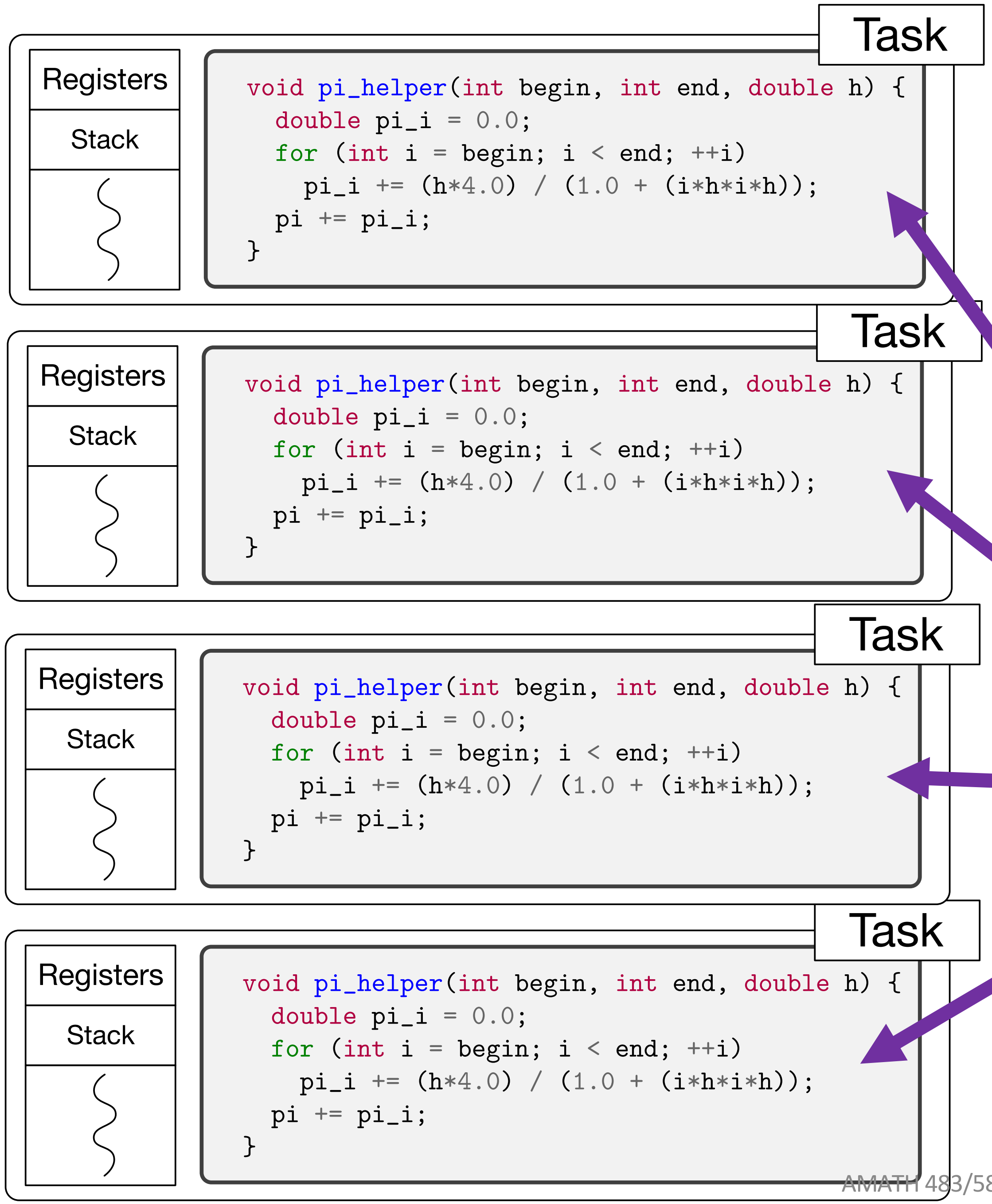
$$\pi_2 = h \sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2}$$

$$\pi_3 = h \sum_{i=3N/4}^{N-1} \frac{4}{1+(ih)^2}$$

```
int main() {
  int N = 1024*1024; double h = 1.0/(double) N;
  double pi_0 = 0.0, pi_1 = 0.0;
  double pi_2 = 0.0, pi_3 = 0.0;

  for (int i = 0; i < N/4; ++i)
    pi_0 += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi_1 += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi_2 += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi_3 += (h*4.0) / (1.0 + (i*h*i*h));

  double pi = pi_0 + pi_1 + pi_2 + pi_3;
  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```




```
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  pi += pi_i;
}

int main(int argc, char* argv[]) {
  int N = 1024*1024; double h = 1.0/(double)N;

  std::thread t0(pi_helper, 0,     N/4,   h);
  std::thread t1(pi_helper, N/4,   N/2,   h);
  std::thread t2(pi_helper, N/2,   3*N/4, h);
  std::thread t3(pi_helper, 3*N/4, N,     h);

  t0.join();  t1.join();  t2.join();  t3.join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```




```cpp
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  pi += pi_i;
}

int main(int argc, char* argv[]) {
  int N = 1024*1024;   int numblocks = 4;
  double h = 1.0/(double)N;

  std::vector<std::thread> threads;
  for (int i = 0; i < numblocks; ++i)
    threads.push_back(
      std::thread(pi_helper, 0,     N/4,   h));

  for (int i = 0; i < numblocks; ++i)
    threads[i].join();

  std::cout << "pi is ~ " << pi << std::endl;

  return 0;
}
```

Task

Registers

Stack

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  pi_mutex.lock();
  pi += pi_i;
  pi_mutex.unlock();
}
```

Task

Registers

Stack

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  pi_mutex.lock();
  pi += pi_i;
  pi_mutex.unlock();
}
```

```
nt main(int argc, char* argv[]) {
int N = 1024*1024;   int numblocks = 4;
double h = 1.0/(double)N;

std::vector<std::thread> threads;
for (int i = 0; i < numblocks; ++i)
  threads.push_back(
    std::thread(pi_helper, 0, N/4, h));

fo  (int i = 0; i < numblocks; ++i)
  threads[i].join();

 td::cout << "pi is ~ " << pi << std::endl;

return 0;
```

Deadlock

Task

Registers

Stack

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  pi_mutex.lock();
  pi += pi_i;
  pi_mutex.unlock();
}
```

What if I return from here without unlock()?

Task

Registers

Stack

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  pi_mutex.lock();
  pi += pi_i;
  pi_mutex.unlock();
}
```

Can never acquire pi_mutex

Deadlock!

Task

Registers

Stack

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  { std::lock_guard<std::mutex> pi_guard(pi_mutex);
    pi += pi_i;
  }
}
```

```
t main(int argc, char* argv[]) {
int N = 1024*1024;   int numblocks = 4;
double h = 1.0/(double)N;

std::vector<std::thread> threads;
for (int i = 0; i < numblocks; ++i)
  threads.push_back(
    std::thread(pi_helper, 0, N/4, h));

for (int i = 0; i < numblocks; ++i)
  threads[i].join();

:cout << "pi is ~ " << pi << std::endl;

eturn 0;
```

Task

Registers

Stack

```
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  { std::lock_guard<std::mutex> pi_guard(pi_mutex);
    pi += pi_i;
  }
}
```

Lock Guard

Task

Registers

Stack

```cpp
double pi = 0.0;
std::mutex pi_mutex;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  { std::lock_guard<std::mutex> pi_guard(pi_mutex);
    pi += pi_i;
  }
}
```

Block scope

Lock acquired at construction

Lock released at destruction

Only lock small update of pi

Using RAII

When pi_guard goes out of scope

Results

```
$ time ./a.out 1000000000 1
pi is ~ 3.14159
2.079u 0.004s 0:02.08 99.5%

$ time ./a.out 1000000000 2
pi is ~ 3.14159
2.062u 0.011s 0:01.04 199.0%

$ time ./a.out 1000000000 4
pi is ~ 3.14159
2.185u 0.009s 0:00.56 389.2%

$ time ./a.out 1000000000 6
pi is ~ 3.14159
0.007s 0:00.55 583.6%

$ time ./a.out 1000000000 8
pi is ~ 3.14159
4.091u 0.012s 0:00.53 773.5%
```

One thread

Sequential time

Two threads

Two times speedup

Four times speedup

Six times usage

Four times speedup

Four times speedup

Eight times usage

CP.4: Think in terms of tasks, rather than threads

- "A thread is an **implementation** concept, a way of thinking about the **machine**. A task is an **application** notion, something you'd like to **do**, preferably concurrently with other tasks. Application concepts are easier to reason ab
- "What" (tasks)
- vs "How" (threads)

```cpp
#include <iostream>
#include <future>

void sayHello() {
  std::cout << "Hello World!" << std::endl;
}

int main() {

  std::async(sayHello);
  std::cout << "Task Launched" << std::endl;

  return 0;
}
```

Need for async()

Task

Run task (asynchronously)

Tasks (Policy)

- Launch a task with async
- Pass arguments to a task
- Futures, returning values from a task
- get() and wait()
- Local variables – thread/task scope
- Shared variables – outer scope

- When is a task evaluated?

std::async() and std::future<>

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

                int argc, char *argv[])

  unsigned long intervals = 1024*1024;
  double h = 1.0 / (double) intervals;

  std::future<double> ppi = std::async(partial_pi, 0, intervals, h);

  std::cout << "partial pi is " << h*ppi.get() << std::endl;

  return 0;
}
```

The template argument is the type of the ”IOU”

async() returns an std::future<>

Launch task

Launch task

Cash in ”IOU”

Arguments to task

Numerical Quadrature (Tasks)

```
int main(int argc, char *argv[])
{
  unsigned long intervals  = 1024*1024, num_blocks  = 128, blocksize = intervals / num_blocks;
  double h = 1.0 / (double) intervals;

  std::vector<std::future<double> > partial_sums;

  for (unsigned long k = 0; k < num_blocks; ++k) {
    partial_sums.push_back(std::async(partial_pi, k*blocksize, (k+1)*blocksize, h));
  }

  double pi = 0.0;
  for (unsigned long k = 0; k < num_blocks; ++k) {
    pi += h*partial_sums[k].get();
  }
  std::cout << "pi is approximately " << pi << std::endl;

  return 0;
}
```

Promise a double

Vector of futures

Launch tasks: each computes a partial sum

Cash in the IOUs

Numerical Quadrature Task

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

Nothing remarkable about this function

Nothing remarkable about this function

Performance

```
$ time ./taskpi 500000000 1
pi is approximately 3.14159
2.006u 0.006s 0:02.01 99.5%
```

CPU time | OS time | Elapsed time | Utilization

```
$ time ./taskpi 500000000 2
pi is approximately 3.14159
1.895u 0.008s 0:00.95 198.9%
```

CPU time | OS time | Elapsed time | Utilization

```
$ time ./taskpi 500000000 4
pi is approximately 3.14159
2.020u 0.007s 0:00.51 396.0%
```

CPU time | OS time | Elapsed time | Utilization

Performance

```
$ time ./taskpi 500000000 1
pi is approximately 3.
2.006u 0.006s 0:02.01 99.5%
```

```
$ time ./taskpi 500000000 2
pi is approximately 3.14159
1.895u 0.008s 0:00.95 198.0%
```

```
$ time ./taskpi 500000000 4
pi is approximately 3.14159
2.020u 0.007s 0:00.51 396.
```

Elapsed time | Utilization | OS time | CPU time

```
$ time ./taskpi 500000000 8
pi is approximately 3.14159
3.669u 0.008s 0:00.48 762.5%
```

Elapsed time | Utilization | OS time | CPU time

```
$ time ./taskpi 500000000 16
pi is approximately 3.14159
3.659u 0.009s 0:00.48 760.4%
```

Elapsed time | Utilization | OS time | CPU time

```
$ time ./taskpi 500000000 50000
pi is approximately 3.14159
2.963u 1.194s 0:00.92 451.0%
```

Too many threads

Asynchrony != Parallelism

Launch async task

When does it run?

Asynchronously

```
int main(int argc, char* argv[]) {
  unsigned long intervals   = 1024 * 1024;
  unsigned long num_blocks  = 1;
  double        h           = 1.0 / (double)intervals;
  unsigned long blocksize   = intervals / num_blocks;

  std::vector<std::future<double>> partial_sums;

  for (unsigned long k = 0; k < num_blocks; ++k)
    partial_sums.push_back(
      std::async(partial_pi, k * blocksize, (k + 1) * blocksize, h));

  for (unsigned long k = 0; k < num_blocks; ++k)
    pi += h * partial_sums[k].get();

  std::cout << "pi is approximately " << pi << std::endl;

  return 0;
}
```

Results

```
time ./a.out 1000000000 1
pi is approximately 3.14159265458974
2.131u 0.006s 0:02.14 99.5%

time ./a.out 1000000000 2
pi is approximately 3.14159265458986
2.118u 0.005s 0:02.12 99.5%

time ./a.out 1000000000 4
pi is approximately 3.14159265458984
2.104u 0.005s 0:02.11 99.5%
```

No speedup!

No speedup!

Launching async()

```
int main(int argc, char* argv[]) {
  unsigned long intervals   = 1024 * 1024;
  unsigned long num_blocks  = 1;
  double        h           = 1.0 / (double)intervals;
  unsigned long blocksize   = intervals / num_blocks;

  std::vector<std::future<double>> partial_sums;

  for (unsigned long k = 0; k < num_blocks; ++k)
    partial_sums.push_back(
      std::async(std::launch::deferred,
        partial_pi, k * blocksize, (k + 1) * blocksize, h));

  for (unsigned long k = 0; k < num_blocks; ++k)
    pi += h * partial_sums[k].get();

  std::cout << "pi is approximately " << pi << std::endl;

  return 0;
}
```

Don't run right away

Not run until here in fact

Results

```
time ./a.out 1000000000 1
pi is approximately 3.14159265458974
2.131u 0.006s 0:02.14 99.5%

time ./a.out 1000000000 2
pi is approximately 3.14159265458986
2.118u 0.005s 0:02.12 99.5%

time ./a.out 1000000000 4
pi is approximately 3.14159265458984
2.104u 0.005s 0:02.11 99.5%
```

No speedup!

No speedup!

Launching async()

```
int main(int argc, char* argv[]) {
  unsigned long intervals   = 1024 * 1024;
  unsigned long num_blocks  = 1;
  double        h           = 1.0 / (double)intervals;
  unsigned long blocksize   = intervals / num_blocks;

  std::vector<std::future<double>> partial_sums;

  for (unsigned long k = 0; k < num_blocks; ++k)
    partial_sums.push_back(
      std::async(std::launch::async,
        partial_pi, k * blocksize, (k + 1) * blocksize, h));

  for (unsigned long k = 0; k < num_blocks; ++k)
    pi += h * partial_sums[k].get();

  std::cout << "pi is approximately " << pi << std::endl;

  return 0;
}
```

Run right away

Results will be here

Results

Speedup!

Speedup!

```
$ time ./a.out 1000000000 1
pi is approximately 3.14159265458974
2.102u 0.011s 0:02.12 99.5%

$ time ./a.out 1000000000 2
pi is approximately 3.14159265458986
2.024u 0.011s 0:01.02 199.0%

$ time ./a.out 1000000000 4
pi is approximately 3.14159265458984
2.171u 0.010s 0:00.55 396.3%
```

Launching async()

```
int main(int argc, char* argv[]) {
  unsigned long intervals   = 1024 * 1024;
  unsigned long num_blocks  = 1;
  double        h           = 1.0 / (double)intervals;
  unsigned long blocksize   = intervals / num_blocks;

  std::vector<std::future<double>> partial_sums;

  for (unsigned long k = 0; k < num_blocks; ++k)
    partial_sums.push_back(
      std::async(
        partial_pi, k * blocksize, (k + 1) * blocksize, h));

  for (unsigned long k = 0; k < num_blocks; ++k)
    pi += h * partial_sums[k].get();

  std::cout << "pi is approximately " << pi << std::endl;

  return 0;
}
```

Default runs sometime

Launching async()

```
int main(int argc, char* argv[]) {
  unsigned long intervals   = 1024 * 1024;
  unsigned long num_blocks  = 1;
  double        h           = 1.0 / (double)intervals;
  unsigned long blocksize   = intervals / num_blocks;

  std::vector<std::future<double>> partial_sums;

  for (unsigned long k = 0; k < num_blocks; ++k)
    partial_sums.push_back(
      std::async(std::launch::async | std::launch::deferred,
        partial_pi, k * blocksize, (k + 1) * blocksize, h));

  for (unsigned long k = 0; k < num_blocks; ++k)
    pi += h * partial_sums[k].get();

  std::cout << "pi is approximately " << pi << std::endl;

  return 0;
}
```

Could be either

Best practice: Always specify launch::async

Summary of C++ features

- std::thread, std::thread::join(), std::thread::detach() — Low level
- std::future, std::async — Task based concurrency / parallelism
  - std::future — Hold task return value
  - std::async — Launch asynchronous task
- std::mutex — Low level
- std::lock_guard<T> — Protect code block with RAII
- std::lock — Safely us multiple mutexes
- std::atomic<T> — Atomically work with atomic types

Bonnie and Clyde Redux

```
int bank_balance = 300;

static std::mutex atm_mutex;
static std::mutex msg_mutex;

void withdraw(const string& msg, int amou
  std::lock(atm_mutex, msg_mutex);
  std::lock_guard<std::mutex> message_lock(msg_mutex, std::adopt_lock);

  cout << msg << " withdraws " << to_string(amount) << endl;

  std::lock_guard<std::mutex> account_lock(atm_mutex, std::adopt_lock);

  bank_balance -= amount;
}
```

Mutexes

Avoid deadlock

Lock two mutexes at once

std::atomic

```
atomic<int>        bank_balance(300
static std::mutex msg_mutex;

void withdraw(const string& msg, int amount) {

  { std::lock_guard<std::mutex> message_lock(msg_mutex);
    cout << msg << " withdraws " << to_string(amount) << endl;
  }

  bank_balance -= amount;
}
```

Bank balance is an indivisible type

NB!! no longer equivalent to
bank_balance = bank_balance – amount;

operator+=(), e.g.

Certain operators are guaranteed to be atomic

This is Broken and Still has a Race

```
atomic<int>        bank_balance(300);
static std::mutex msg_mutex;

void withdraw(const string& msg, int amount) {

  { std::lock_guard<std::mutex> message_lock(msg_mutex);
    cout << msg << " withdraws " << to_string(amount) << endl;
  }

  bank_balance = bank_balance - amount;
}
```

Bank balance is an indivisible type

Not atomic!

Only operator-=() is atomic

std::atomic

Bank balance is an indivisible type

```
atomic<int>       bank_balance(300);
static std::mutex msg_mutex;

void withdraw(const string& msg, int amount) {

  { std::lock_guard<std::mutex> message_lock(msg_mutex);
    cout << msg << " withdraws " << to_string(amount) << endl;
  }

  bank_balance -= amount;
}
```

operator+=(), e.g.

Certain operators are guaranteed to be atomic

std::atomic

Can we fix pi with atomic?

```
double pi = 0.0;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  pi += pi_i;
}
```

std::atomic

double is not an integral type

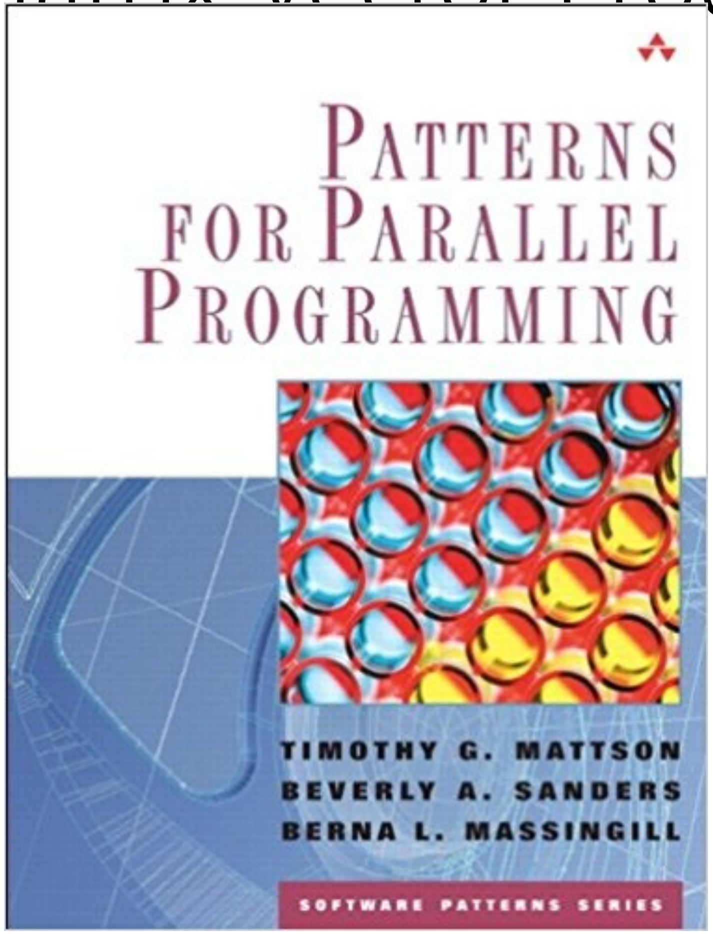
No atomic double!

```
std::atomic<double> pi = 0.0;

void pi_helper(int begin, int end, double h) {
  double pi_i = 0.0;
  for (int i = begin; i < end; ++i)
    pi_i += (h*4.0) / (1.0 + (i*h*i*h));
  pi += pi_i;
}
```

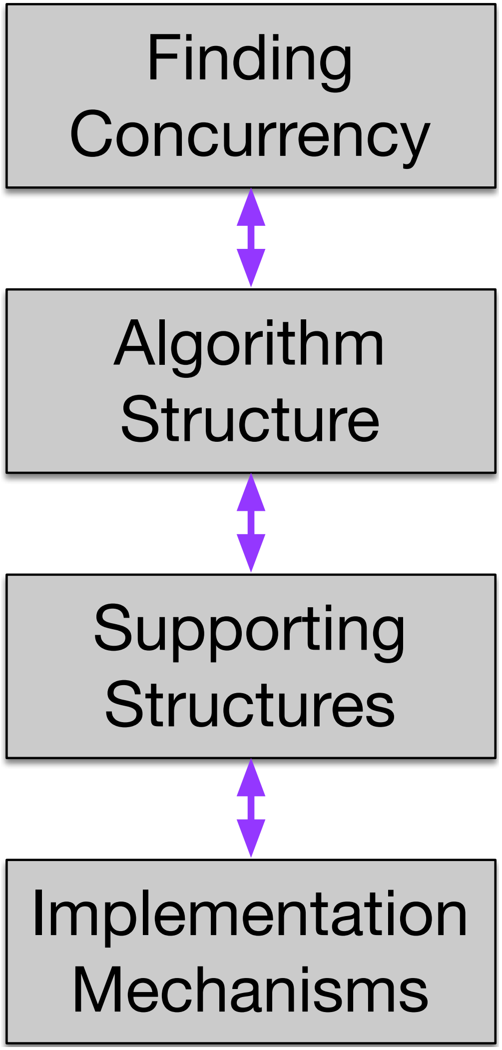
Core Guidelines Rule Summary

- **CP.1: Assume that someone someday will run your code as part of a multi-threaded program**
- **CP.2: Avoid data races**
- **CP.3: Minimize explicit sharing of writable data**
- **CP.4: Think in terms of tasks, rather than threads**
- **CP.9: Whenever feasible use tools to validate concurrent code**
- **CP.20: Use RAII, never plain lock()/unlock()**
- **CP.21: Use std::lock() to acquire multiple mutexes**
- **Use std::launch::async when using std::async()**
- **Use std::atomic<> for updating integral types (carefully!)**

Matrix-Vector Product



Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*(First ed.). Addison-Wesley Professional.



Matrix-Vector Product $y \leftarrow Ax$

$$\forall\, i : \quad y_i = \sum_{i=0}^{i<M} A_{ik} x_k$$

Each summation is independent of i

Make computation of each y(i) a task

```
void matvec(const Matrix& A, const Vector& x, Vector& y) {
  for (int i = 0; i < A.numRows(); ++i) {
    for (int k = 0; k < A.numCols(); ++k) {
      y(i) += A(i, k) * x(k);
    }
  }
}
```

Each inner loop is independent of i

Async Matrix-Vector Product

```
double inner_dot(const Matrix& A, const Vector& x, unsigned long i, double init
  for (unsigned long j = 0; j < A.numCols(); ++j) {
    init += A(i, j) * x(j);
  }
  return init;
}


void task_matvec(const Matrix& A, const Vector& x, Vector& y) {
  std::vector<std::future <double> > futs(A.numRows());
  for (int i = 0; i < A.numRows(); ++i) {
    futs[i] = std::async(inner_dot, A, x, i, 0.0);
  }
  for (int i = 0; i < A.numRows(); ++i) {
    y(i) = futs[i].get();
  }
}
```

Row times column

Make computation of each y(i) an asynchronous task

Cash in IOU

Results

```
$ time ./task_matvec
1.798u 3.544s 0:05.32 100.1%    0+0k 0+0io 0pf+0w
```

User time

System time

Bad!

Partitioned Matrix-Vector Product

Return a Vector

```
Vector inner_dot(const Matrix& A, const Vector& x, unsigned long begin, ur
  Vector z(end-begin, 0);
  for (unsigned long i = 0; i < end-begin; ++i) {
    for (unsigned long j = 0; j < A.numCols(); ++j) {
      z(i) += A(i+begin, j) * x(j);
    }
  }
  return z;
}
```

Do a range of rows

Row times column

Results

```
$ time ./task_matvec_2 8192 1
0.922u 0.357s 0:01.28 99.2%

$ time ./task_matvec_2 8192 2
1.078u 0.529s 0:01.55 102.5%

$ time ./task_matvec_2 8192 4
1.357u 0.876s 0:02.15 103.2%

$ time ./task_matvec_2 8192 8
1.936u 1.575s 0:03.42 102.3%
```

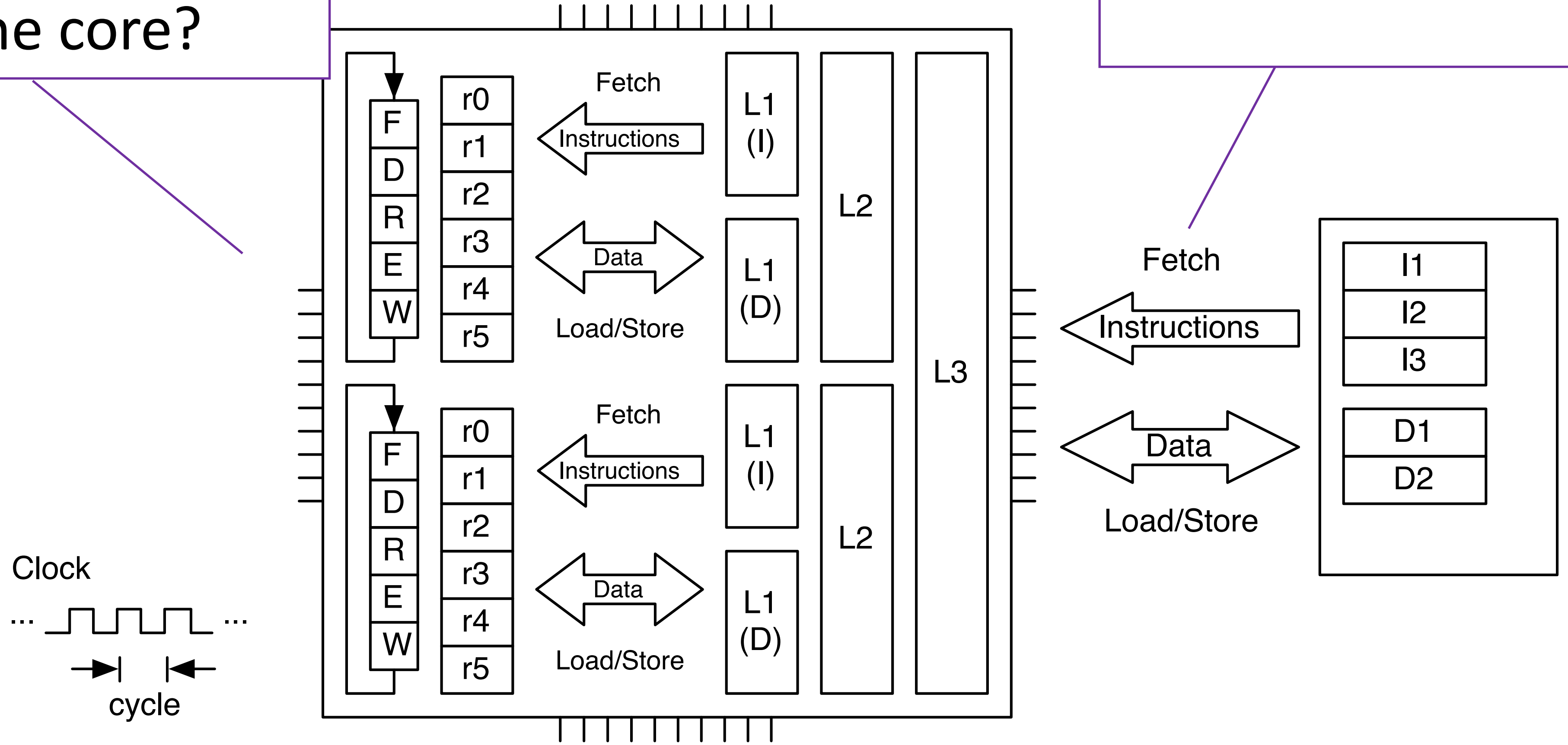
What ***might*** be happening?

Not much speedup

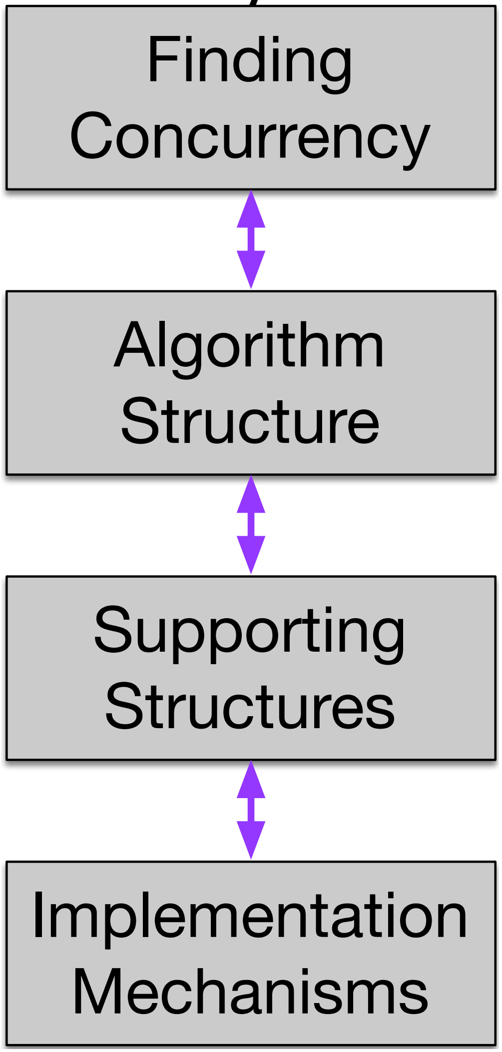
What's Wrong?

What was the bottleneck in matvec on one core?

If one core can't get data fast enough through here, will two cores do better?



Asynchronous Matrix-Matrix Product



$$\forall\, i, j : \quad C_{i,j} = \sum_{k=0}^{k<M} A_{ik} B_{kj}$$

Each summation is independent of i,j

$$\forall\, I, J : \quad C_{I,J} = \sum_{K=0}^{K<M} A_{IK} B_{KJ}$$

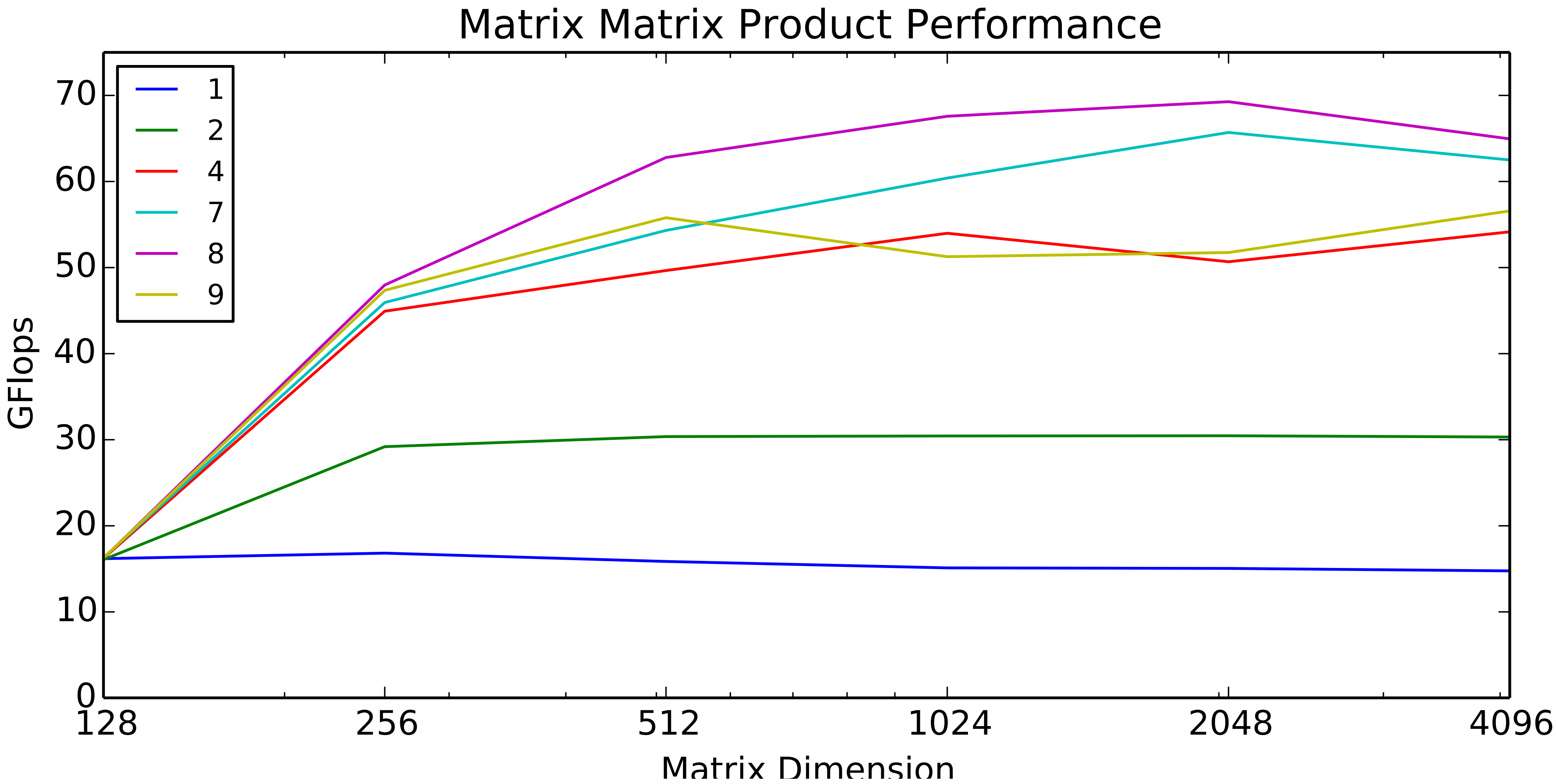
Also true if A, B, and C are blocks

Matrix-Matrix

Make this a task

```
for (int ii = 0; ii < A.numRows(); ii += blocksize) {
  for (int jj = 0; jj < B.numCols(); jj += blocksize) {
    for (int kk = 0; kk < A.numCols(); kk += blocksize) {

      for (int i = ii; i < ii+blocksize; i += 2) {
        for (int j = jj; j < jj+blocksize; j += 2) {

          double t00 = C(i,j);        double t01 = C(i,j+1);
          double t10 = C(i+1,j);      double t11 = C(i+1,j+1);

          for (int k = kk; k < kk+blocksize; ++k) {
            t00 += A(i  , k) * B(k, j  );
            t01 += A(i  , k) * B(k, j+1);
            t10 += A(i+1, k) * B(k, j  );
            t11 += A(i+1, k) * B(k, j+1);
          }

          C(i,   j) = t00;  C(i,   j+1) = t01;
          C(i+1,j) = t10;  C(i+1,j+1) = t11;

        }
      }
    }
  }
}
```

Asynchronous Matrix-Matrix Product



Thank You!