Threads & Synchronization

Lecture 11

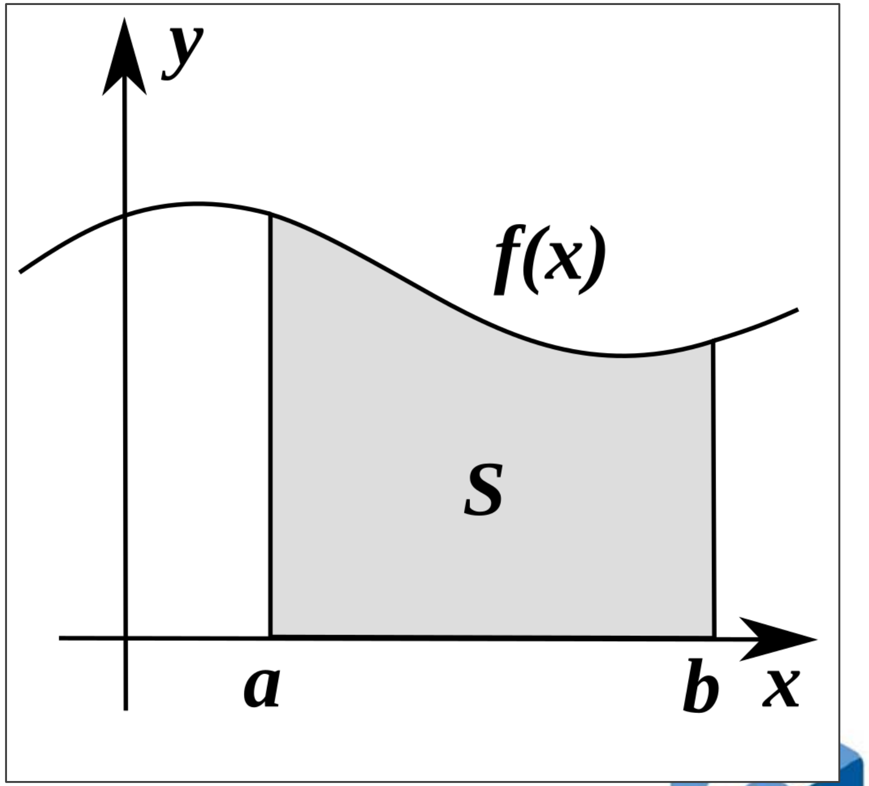
Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4400/

Finding π by Numerical Integration

Numerical Integration

- Numerical techniques for computing integrals
  - Variations of Riemann sums
  - We will also look at Simpson's rule

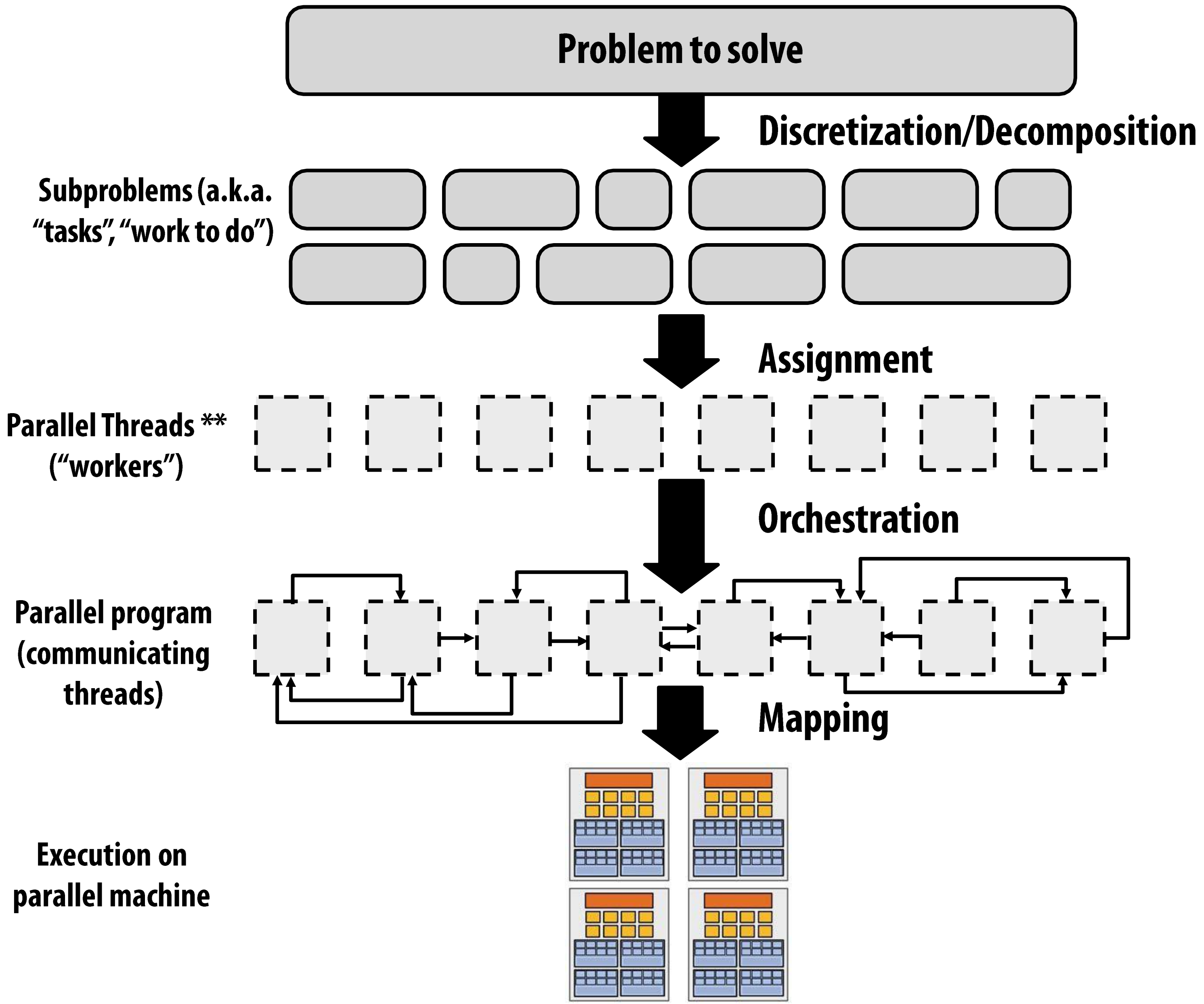


- An definite integral y = $f(x)$

$$\int_a^b f(x)dx$$

- Is defined as the area S under the graph of the function

$$f(x)$$


Problem to solve
Discretization/Decomposition
Subproblems (a.k.a. "tasks", "work to do")
Assignment
Parallel Threads **
("workers")
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine

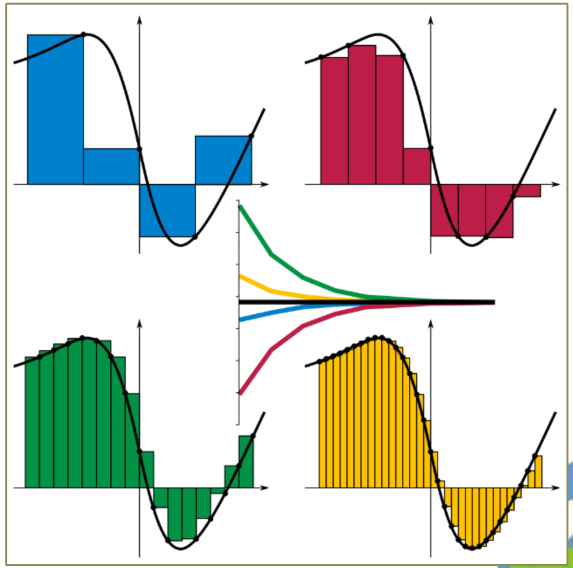
Riemann Sums (Discretization)

- Approximation of an integral using a finite sum, named after the German mathematician Bernhard Riemann (19^{th} century)
- The sum is calculated by partitioning the region into shapes
- Riemann sum is obtained by choosing n points $\{x_j\}$ in the interval $[a, b]$, such that $a = x_0 < x_1 < \cdots < x_n = b$
- And then sum up

$$S_n = \sum_j f(x_j)\Delta x_j, \text{ where } \Delta x_j = x_{j+1} - x_j$$

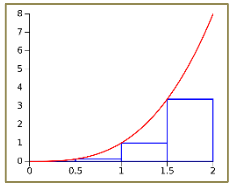
- The integral is then defined as

$$\int_a^b f(x)dx = \lim_{\|\Delta x\| \to 0} \sum_j f(x_j)\Delta x_j$$

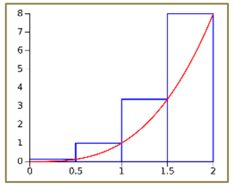


Riemann Sums (Discretization)

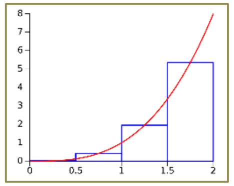
- Different variations
  - Divide $[a, b]$ into n equal sub-intervals of width Δx
  - Then the partition points are: $a, a + \Delta x, a + 2\Delta x, \dots, a + (n-1)\Delta x, b$
- Left rule
  - $S_{left} = \Delta x[f(a) + f(a+\Delta x) + f(a+2\Delta x) + \cdots + f(b-\Delta x)]$



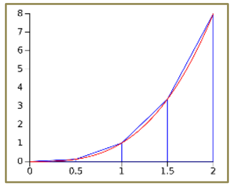
- Right rule
  - $S_{right} = \Delta x[f(a+\Delta x) + f(a+2\Delta x) + \cdots + f(b)]$



- Midpoint rule
  - $S_{mid} = \Delta x[f\left(a+\frac{\Delta x}{2}\right) + f\left(a+3\frac{\Delta x}{2}\right) + \cdots + f\left(b-\frac{\Delta x}{2}\right)]$



- Trapezoidal rule
  - $S_{trap} = \frac{\Delta x}{2}[f(a) + 2f(a+\Delta x) + 2f(a+2\Delta x) + \cdots + f(b)]$



Multi-Dimensional Integrals

- Compute the volume under a given N-dimensional function is similar, except that now the sub-intervals are N-dimensional as well
- The idea is to "break-up" the domain via a partition into pieces
  - then multiply the "size" of each piece by some value the function takes on that piece,
  - and sum all these products.
- This can be generalized to allow Riemann sums for functions over domains of more than one dimension

Exercise

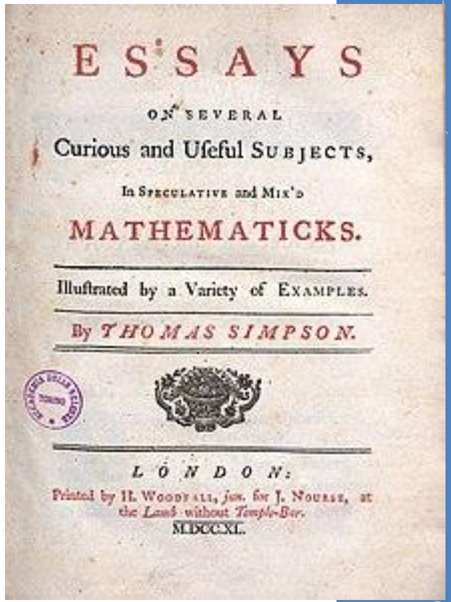
- Compute the left and right Riemann sums for $y = \sin x$, on the intervals $\left[0, \frac{\pi}{2}\right]$ and $\left[\frac{\pi}{2}, \pi\right]$ for increasing numbers of sub-intervals
- Compare the results with the correct results themselves
  - What do you observe?
  - What do you observe if you use the trapezoidal Riemann sum?
  - How many sub-intervals do you need to achieve an accuracy of $eps \leq 10^{-8}$ for the three types of Riemann sums (left, right, trapezoidal)?
- Does it make sense to parallelize the computation of the Riemann sum in terms of gaining a speedup?
  - How many sub-intervals do you need to gain speedup over the sequential case?

Simpson Rule

- Named after the British mathematician Thomas Simpson (18th century)
- Several variations, based on different ranks of the approximations
  - We will focus on Simpsons composite 1/3 rule
  - Relies on approximation with a quadratic function
  - For n sub-intervals we get:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{3} \sum_{i=0}^{\frac{n}{2}} [f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})]$$

$$= \frac{\Delta x}{3} \left[f(x_0) + 4 \sum_{i=0}^{\frac{n}{2}} f(x_{2i-1}) \; + 2 \sum_{i=0}^{\frac{n}{2}-1} f(x_{2i}) + f(x_n) \right]$$



Exercise

- Compute the approximation of the integral for $y = \sin x$, on the intervals $\left[0, \frac{\pi}{2}\right]$ and $\left[\frac{\pi}{2}, \pi\right]$ using the Simpson rule for increasing numbers of sub-intervals
- Compare the results with the correct results themselves
  - What do you observe?
  - How many sub-intervals do you need to achieve an accuracy of $eps \leq 10^{-8}$ for the approximation?
- Does it make sense to parallelize the computation of the Simpson rule in terms of gaining a speedup?
  - How many sub-intervals do you need to gain speedup over the sequential case?

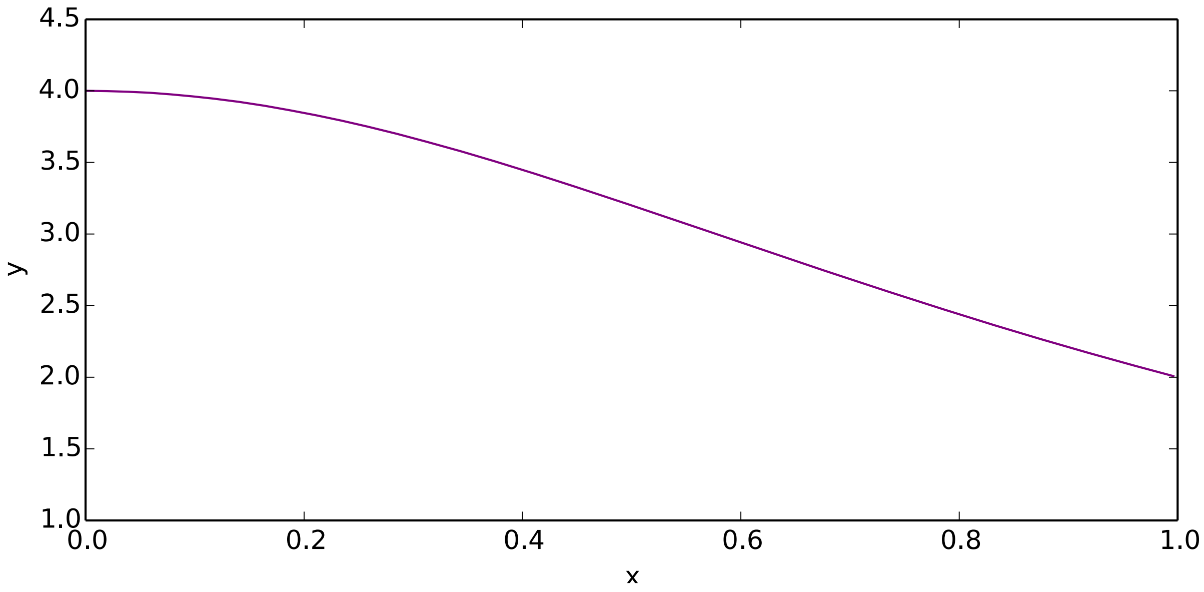
Finding π

- Find the value of

$$\pi$$

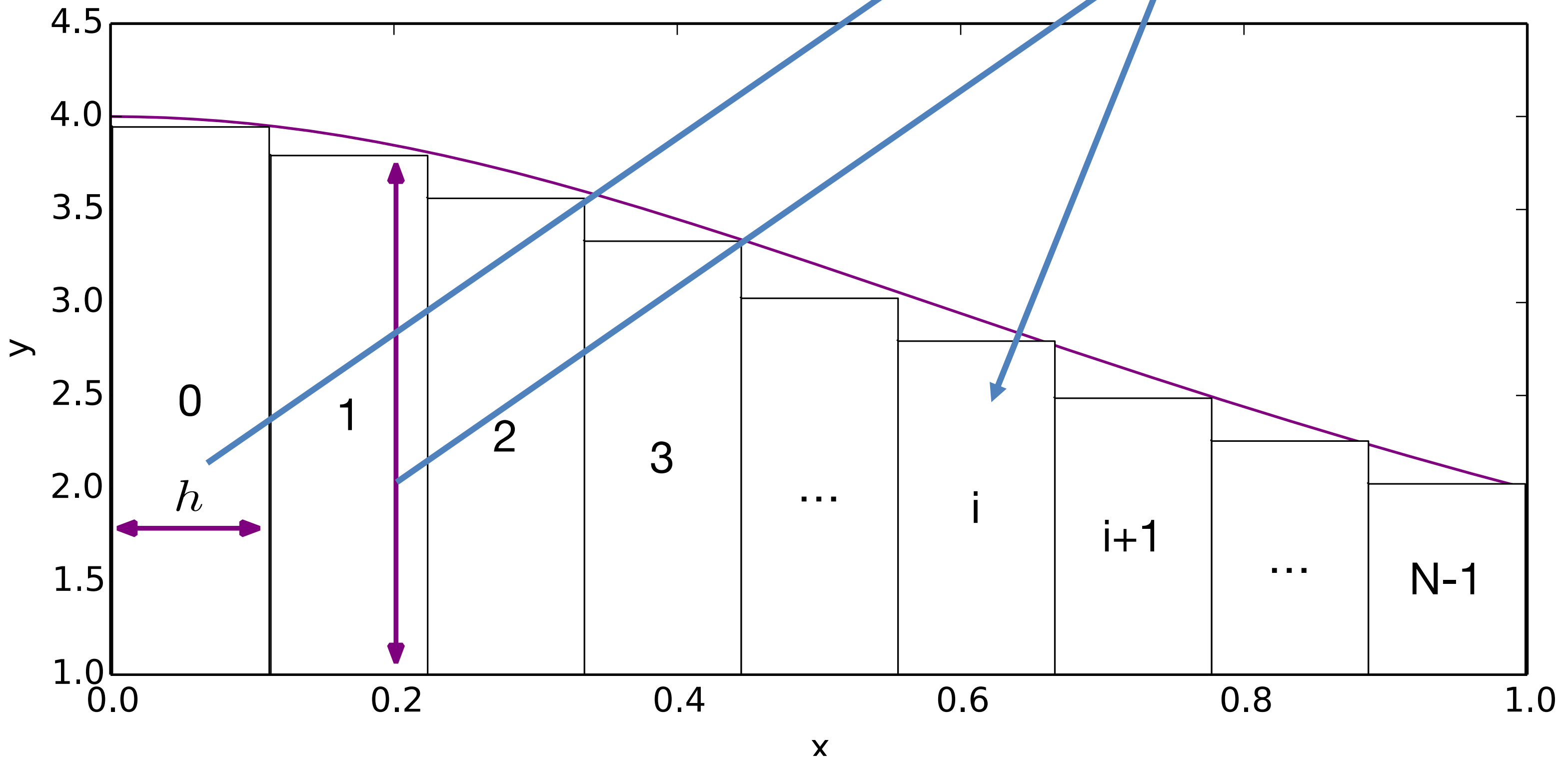
- Using formula

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



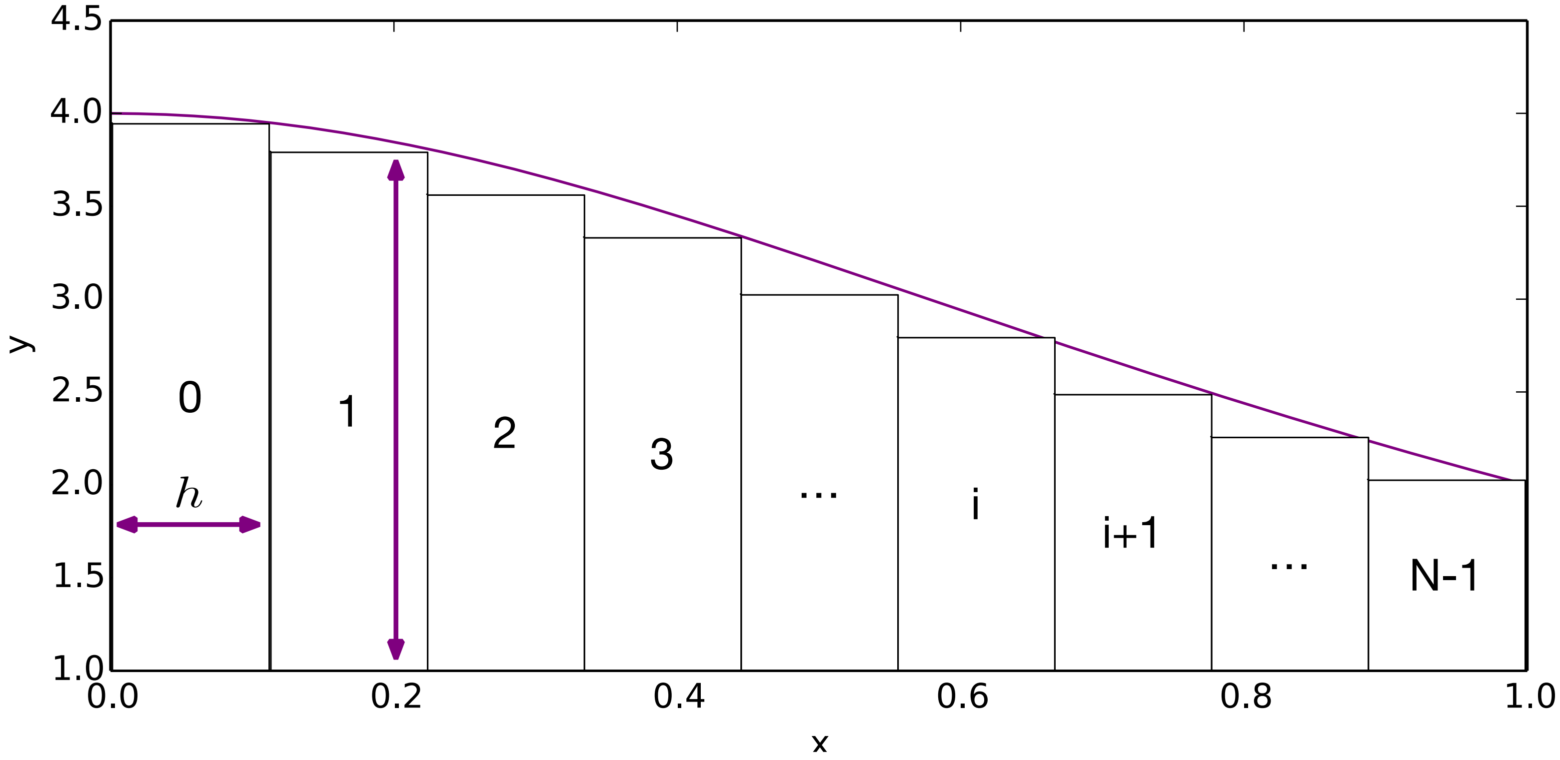
Numerical Integration

$$A = h \frac{4}{1 + (ih)^2}$$



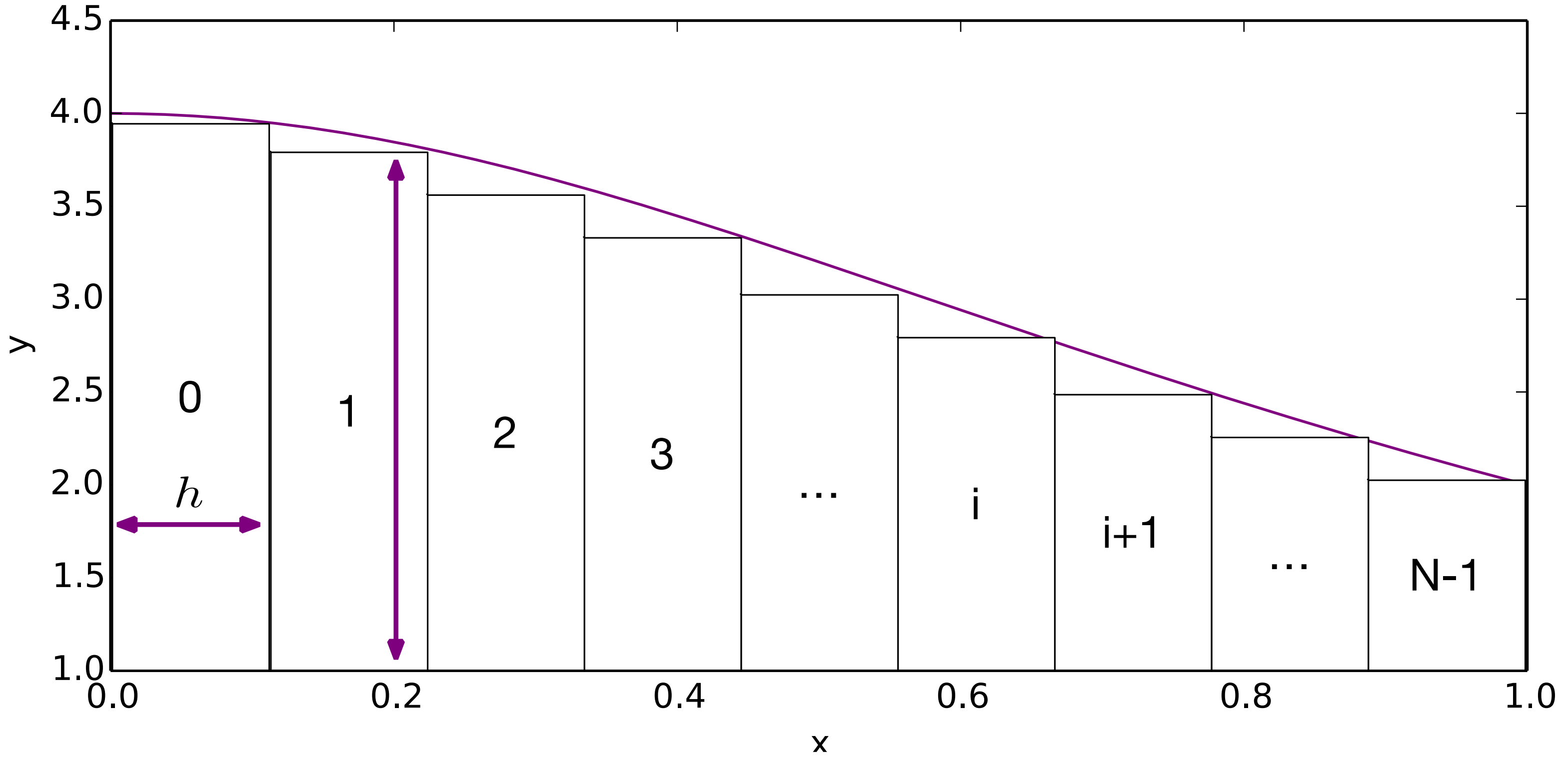
Numerical Integration

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



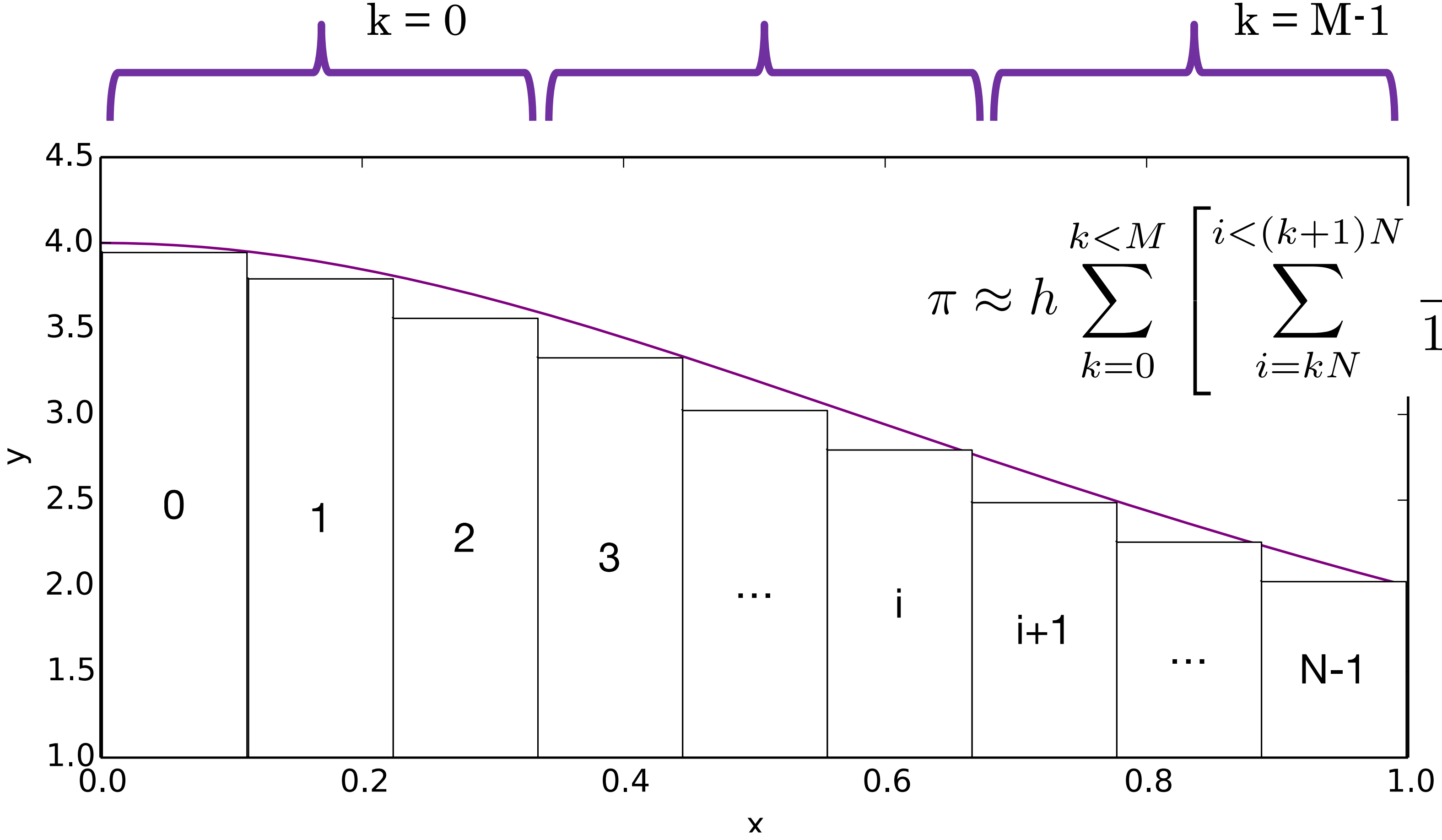
Numerical Integration (Sequential)

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



Finding Concurrency (Decomposition)

Partial sums are all independent   Can be computed concurrently



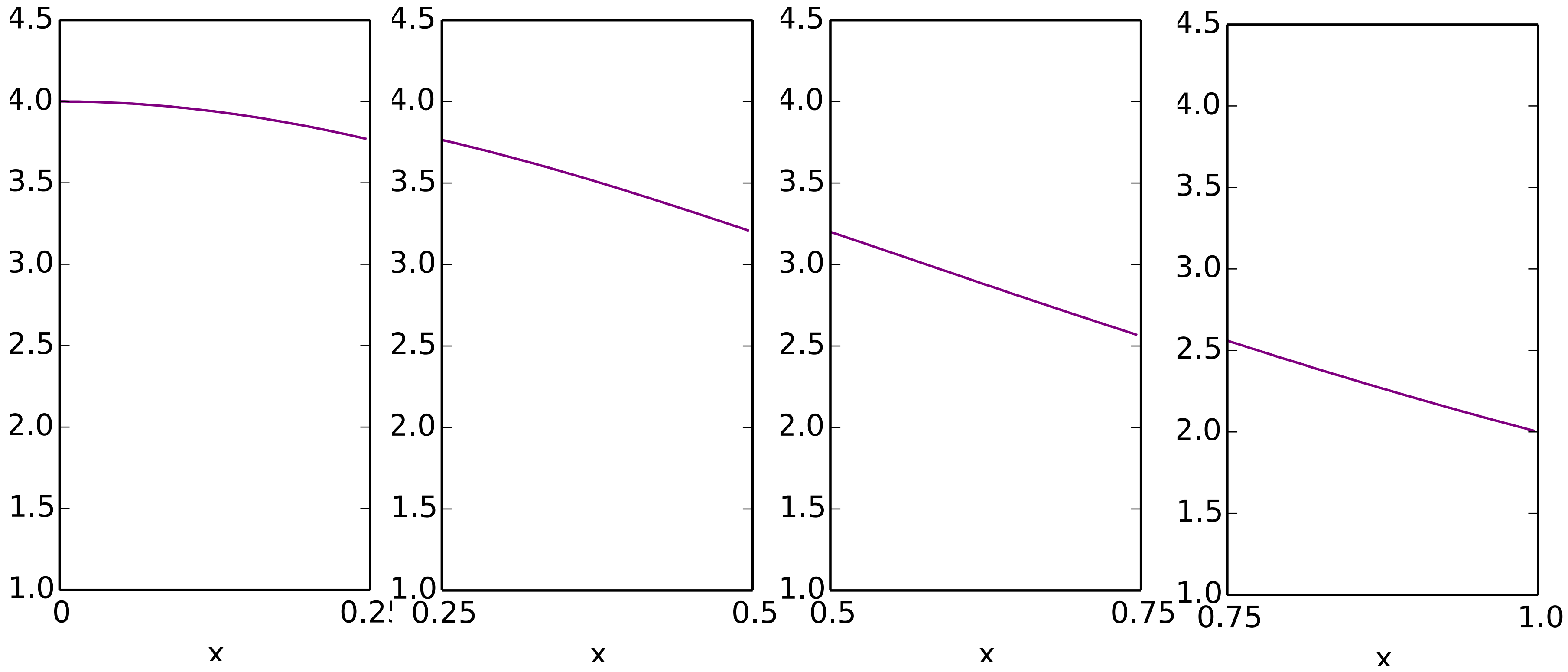
$$\pi \approx h \sum_{k=0}^{k<M} \left[\sum_{i=kN}^{i<(k+1)N} \frac{4}{1+(ih)^2} \right]$$

Sequentially Finding π (Two Nested Loops)

```cpp
double calculate_pi()
{
    double h = 1.0 / (double) num_intervals; // Decomposition
    double pi = 0.0;

    // For each set of discretized points
    for (int k = 0; k < num_intervals; ++k) {
        double partial_pi = 0.0;                // Compute partial sum
        for (int i = k; i < (k + blocksize); ++i)
            partial_pi += 4.0 / (1.0 + sqr(i * h));
        pi += partial_pi;                       // Accumulate final sum
    }
    return pi;
}
```

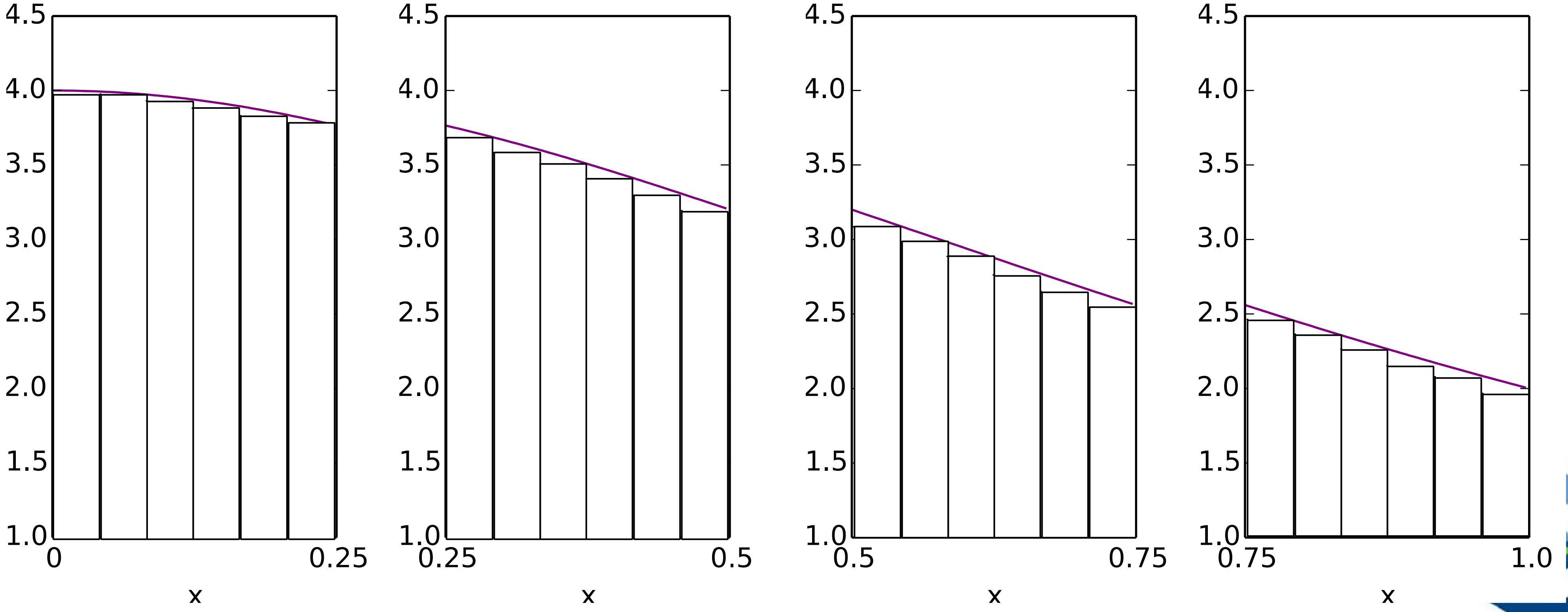
Finding Concurrency (Decomposition)

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



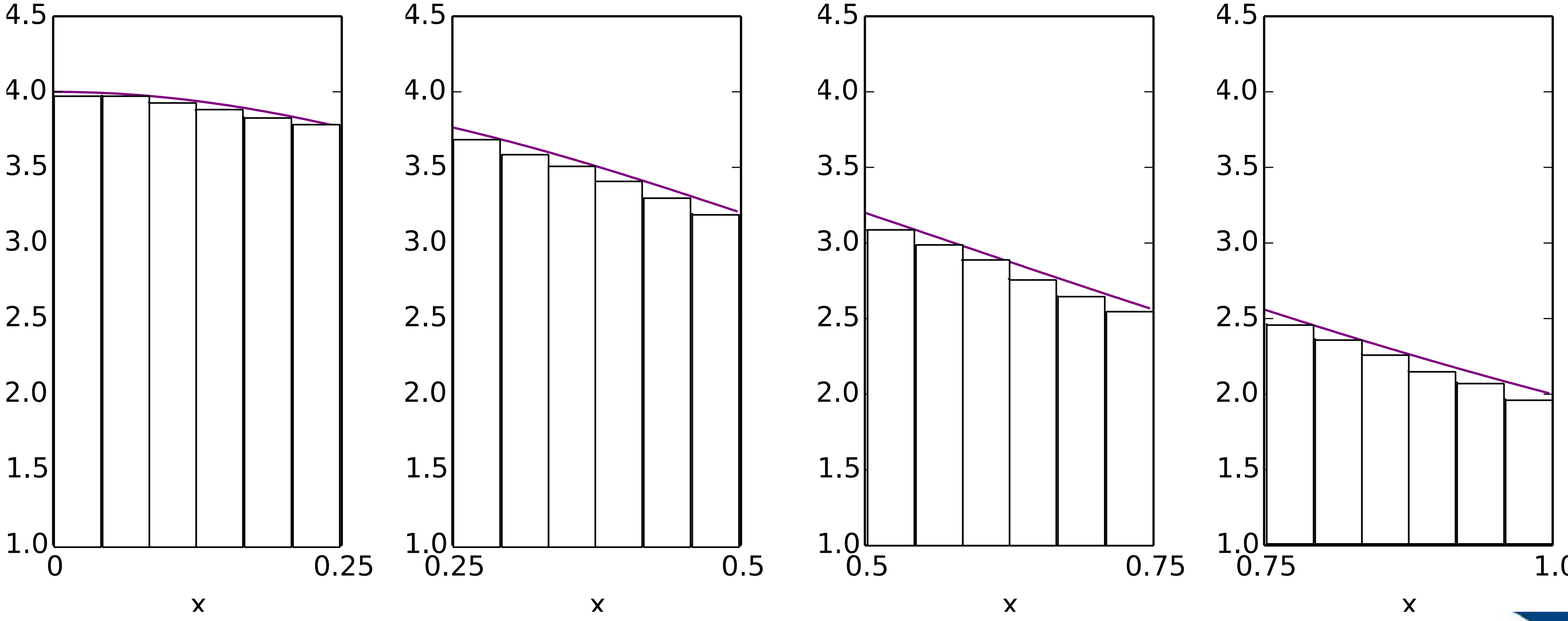
Finding Concurrency (Decomposition)

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



Finding Concurrency (Decomposition)

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



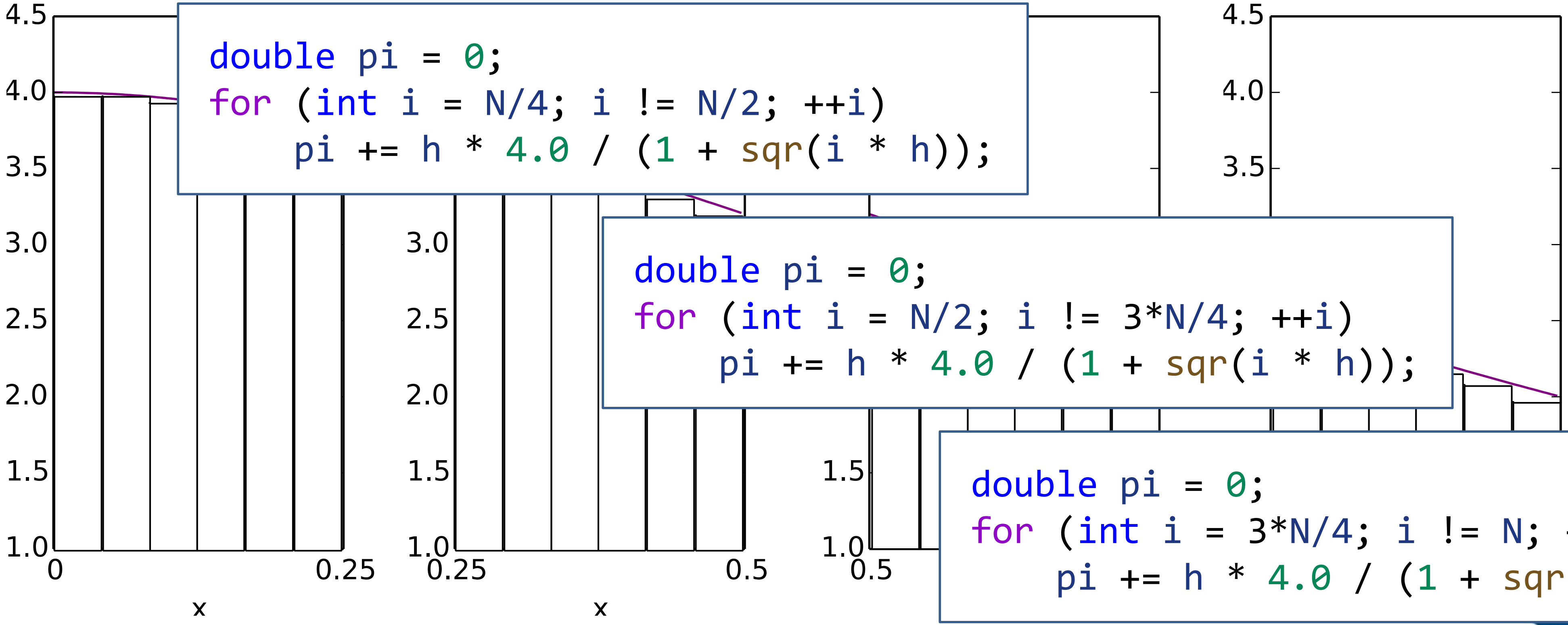
Finding Concurrency (Decomposition)

```
double pi = 0;
for (int i = 0; i != N/4; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```

```
double pi = 0;
for (int i = N/4; i != N/2; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```

```
double pi = 0;
for (int i = N/2; i != 3*N/4; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```

```
double pi = 0;
for (int i = 3*N/4; i != N; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```



Finding Concurrency (Decomposition)

```
int main()
{
    int const N = 1'000'000;
    double pi = 0;

    for (int i = 0; i != N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/4; i != N/2; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/2; i != 3*N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = 3*N/4; i != N; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

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

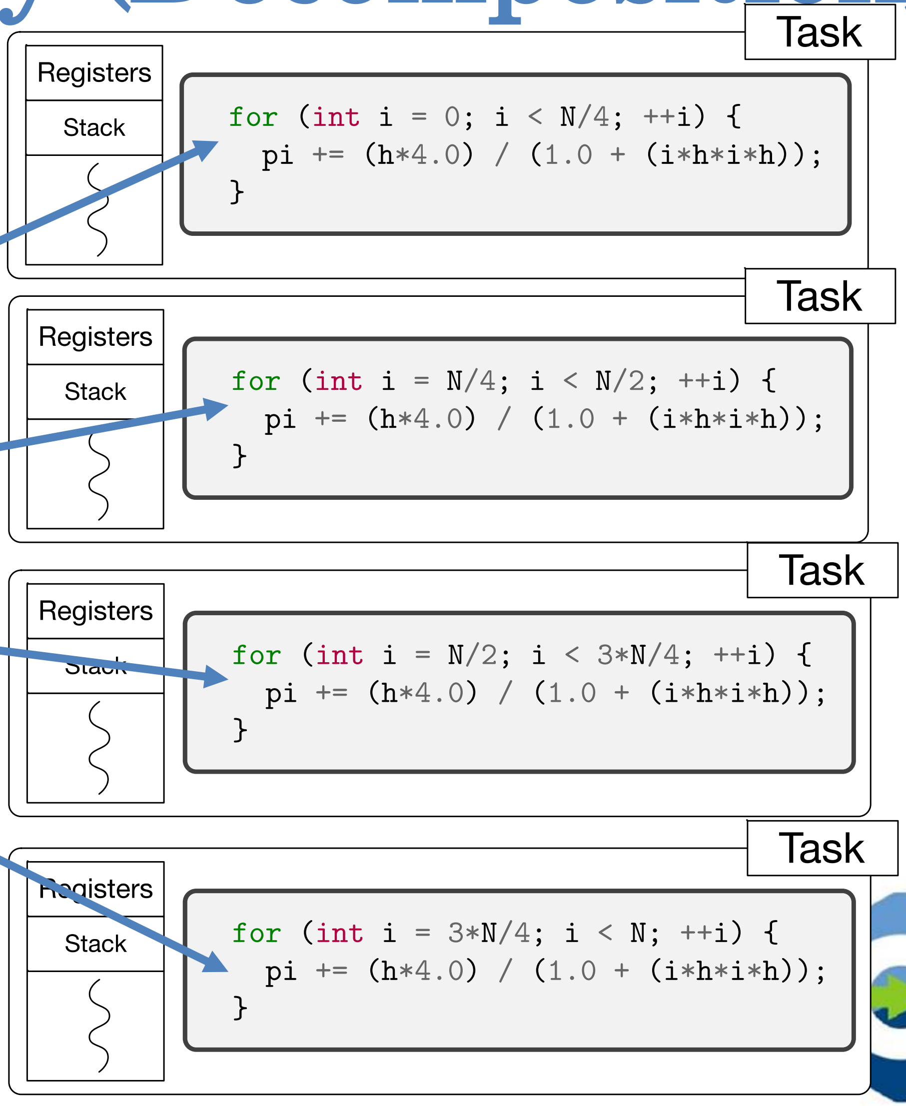
$$\boxed{h\sum_{i=N/4}^{N/2-1}\frac{4}{1+(ih)^2}}$$

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

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

Finding Concurrency (Decomposition)

```cpp
int main()
{
    int const N = 1'000'000;
    double pi = 0;

    for (int i = 0; i != N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/4; i != N/2; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/2; i != 3*N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = 3*N/4; i != N; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

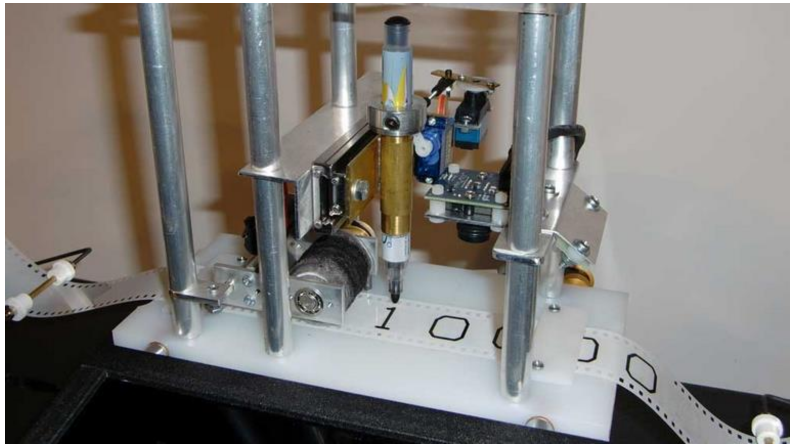


Aside: Threads in C++

What's a 'Thread'?

- C++ Standard defines it as follows:
  - Thread of execution: "single flow of control within a program" ([intro.multithread.general])
  - `std::thread`: is specified as a component "that can be used to create and manage threads" (see [thread.threads.general]), where "thread" explicitly refers to the definition of "threads of execution" (see [thread.thread.class.general])
    - Note: "These threads are intended to map one-to-one with operating system threads"
  - Execution agent: In [thread.req.lockable.general], an execution agent is defined as "an entity such as a thread that may perform work in parallel with other execution agents".

What’s a ‘Thread’?



- An entity which exposes 4 properties:
  - A single flow of control (sequence of op-codes)
  - A program counter marking what’s currently being executed
  - An associated execution context (stack, register set, static and dynamic memory, thread local variables, etc.)
  - A state (initialized, pending, suspended, terminated, etc.)

Library API for Threads

```
template <typename F, typename... Args>
std::thread t(F f, Args... args)
```

- `t` is a new instance representing a thread executing `f` with `args` as its arguments
- Return of `f` is implicitly exiting (terminating) thread

```
template <typename F, typename... Args>
std::jthread t(F f, Args... args)
```

- `t` is a new instance representing a thread executing `f` with `args` as its arguments
- Return of `f` is implicitly exiting (terminating) thread
- Destructor of thread object `t` joins implicitly

```
void (j)thread::join();
```

- Suspends execution of the calling thread until the target thread terminates
- Does nothing if target thread has already finished running

Threads Example

```cpp
int shared_int = 4700;

void thread_func(int num_thread) {
    std::println("Thread {}, stack {}, shared {} ({})", num_thread,
        &num_thread, &shared_int, ++shared_int);
}

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

    std::jthread t1(thread_func, 0));
    std::jthread t2(thread_func, 1));
    std::jthread t3(thread_func, 2));
    std::jthread t4(thread_func, 3));

    return 0;    // destructors of threads will join implicitly
}
```

```
Thread 3, stack 0x59c01ffae0, shared 0x7ff606ee8000 (4701)
Thread 0, stack 0x59bfeff840, shared 0x7ff606ee8000 (4703)
Thread 2, stack 0x59c00ff700, shared 0x7ff606ee8000 (4702)
Thread 1, stack 0x59bffffb10, shared 0x7ff606ee8000 (4704)
```

Threads Example

- How many threads are in this program?
- Does the main thread join with the threads in the same order that they were created?
- Do the threads exit in the same order they were created?
- If we run the program again, would the result change?

Threads vs. Tasks

```
// Task
void say_hello(int thread_num) {
    std::println("Hello World. I am thread {}", thread_num);
}

int main()
{
    std::thread tid[16];
    for (int i = 0; i < 16; ++i)     // Launch threads ('fork')
        tid[i] = std::thread(say_hello, i);

    for (int i = 0; i < 16; ++i)     // Wait for tasks to finish
        tid[i].join();
}
```

Threads vs. Tasks

```
// Task
void say_hello(int thread_num) {
    std::println("Hello World. I am thread {}", thread_num);
}

int main()
{
    std::jthread tid[16];
    for (int i = 0; i < 16; ++i)     // Launch threads ('fork')
        tid[i] = std::jthread(say_hello, i);

}   // implicitly join in destructor of thread objects
```

Threads

```
// Note: thread functions return void
void calculate_partial_pi(long begin, long end) {
    double partial_pi = 0.0;
    for (long i = begin; i < end; ++i)
        partial_pi += 4.0 / (1.0 + sqr(i * h));
    return partial_pi;      // oops!
}

int main() {
    double h = 1.0 / (double) num_intervals;    // Decomposition
    double pi = 0.0;

    for (int k = 0; k < num_intervals; k += blocksize) {
        // how do we get the partial sums here?
        pi += h * partial_pi;
    }

    std::println("pi is approximately {}", pi);
}
```

Threads

```
void calculate_partial_pi(long begin, long end, double h, double& pi) {
    double partial_pi = 0.0;
    for (long i = begin; i < end; ++i)
        partial_pi += 4.0 / (1.0 + sqr(i * h));
    pi += partial_pi;
}

int main() {
    double h = 1.0 / (double) num_intervals;     // Decomposition
    double pi = 0.0;

    std::vector<std::thread> threads;
    for (int k = 0; k < num_blocks; ++k) {
        threads.push_back(
            std::thread(calculate_partial_pi,
                k * block_size, (k + 1) * block_size, h, std::ref(pi));
    }
    for (auto& t : threads) t.join();            // Wait for tasks to finish

    std::println("pi is approximately {}", pi);
}
```

Needs explicit reference!

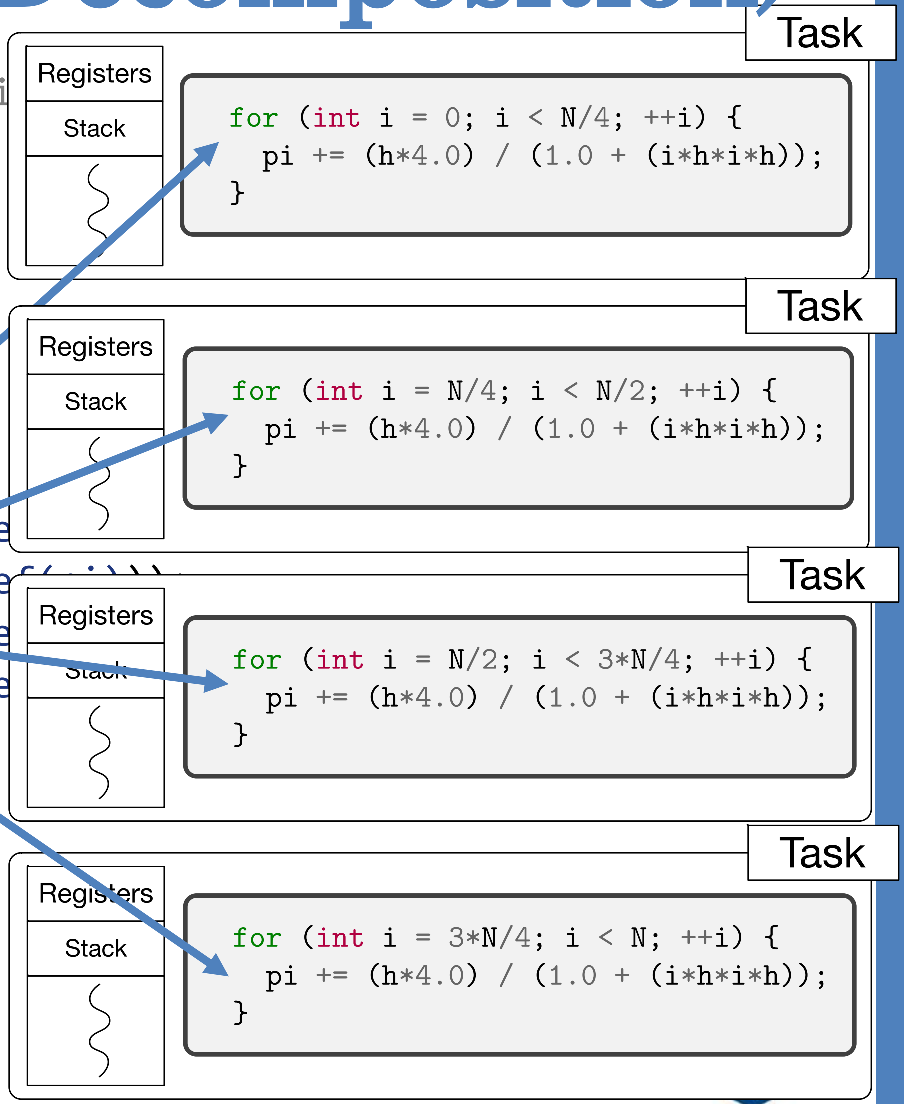
Threads

```
void calculate_partial_pi(long begin, long end, double h, double& pi)
{
    double partial_pi = 0.0;
    for (long i = begin; i < end; ++i)
        partial_pi += 4.0 / (1.0 + sqr(i * h));
    pi += partial_pi;
}
```

- `double partial_pi`: local variable
- `double& pi`: shared variable
- `pi += partial_pi`: update shared variable!

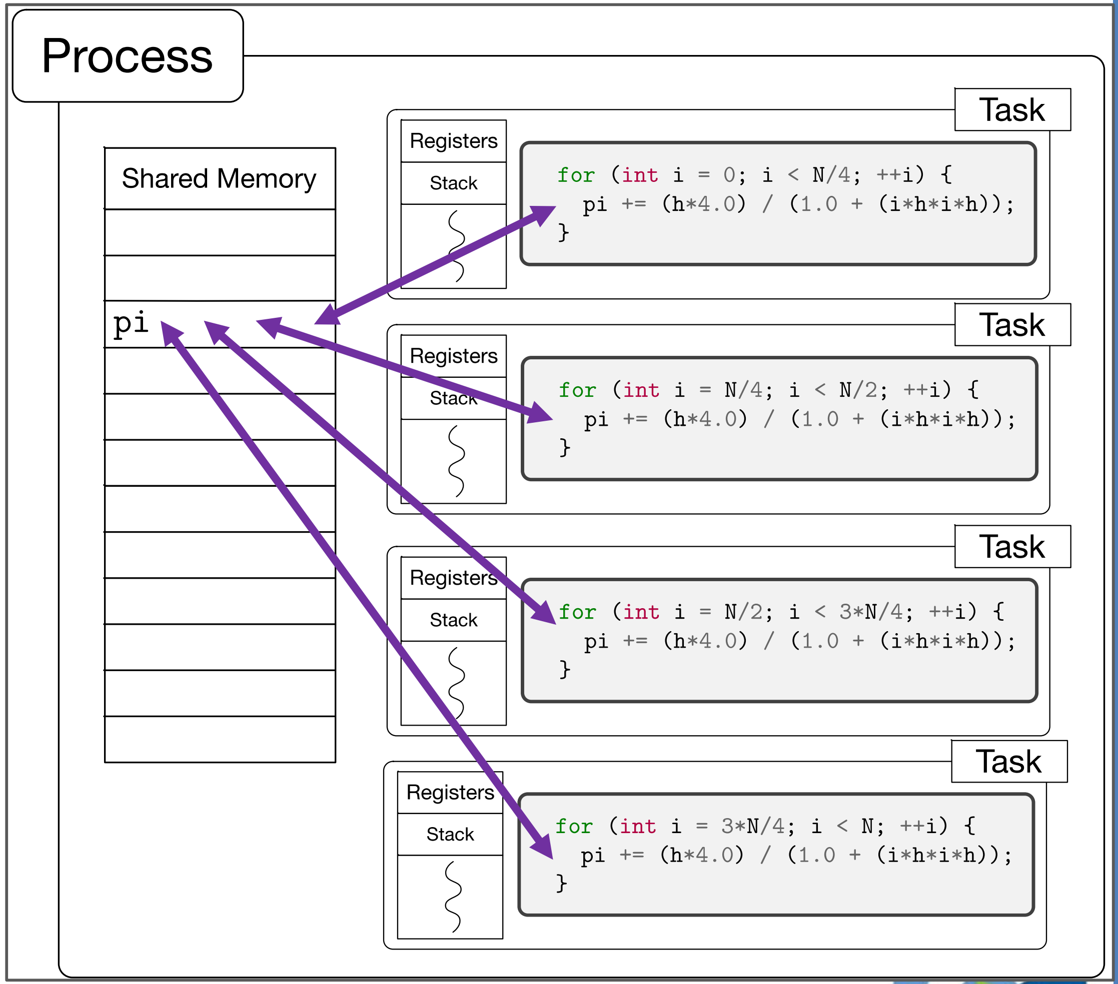
Finding Concurrency (Decomposition)

```
void pi_helper(int begin, int end, double h, double& pi
    for (int i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
}

int main() {
    int const N = 1'000'000;
    double h = 1.0 / N;

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

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

    std::println("pi: {}", pi);
}
```

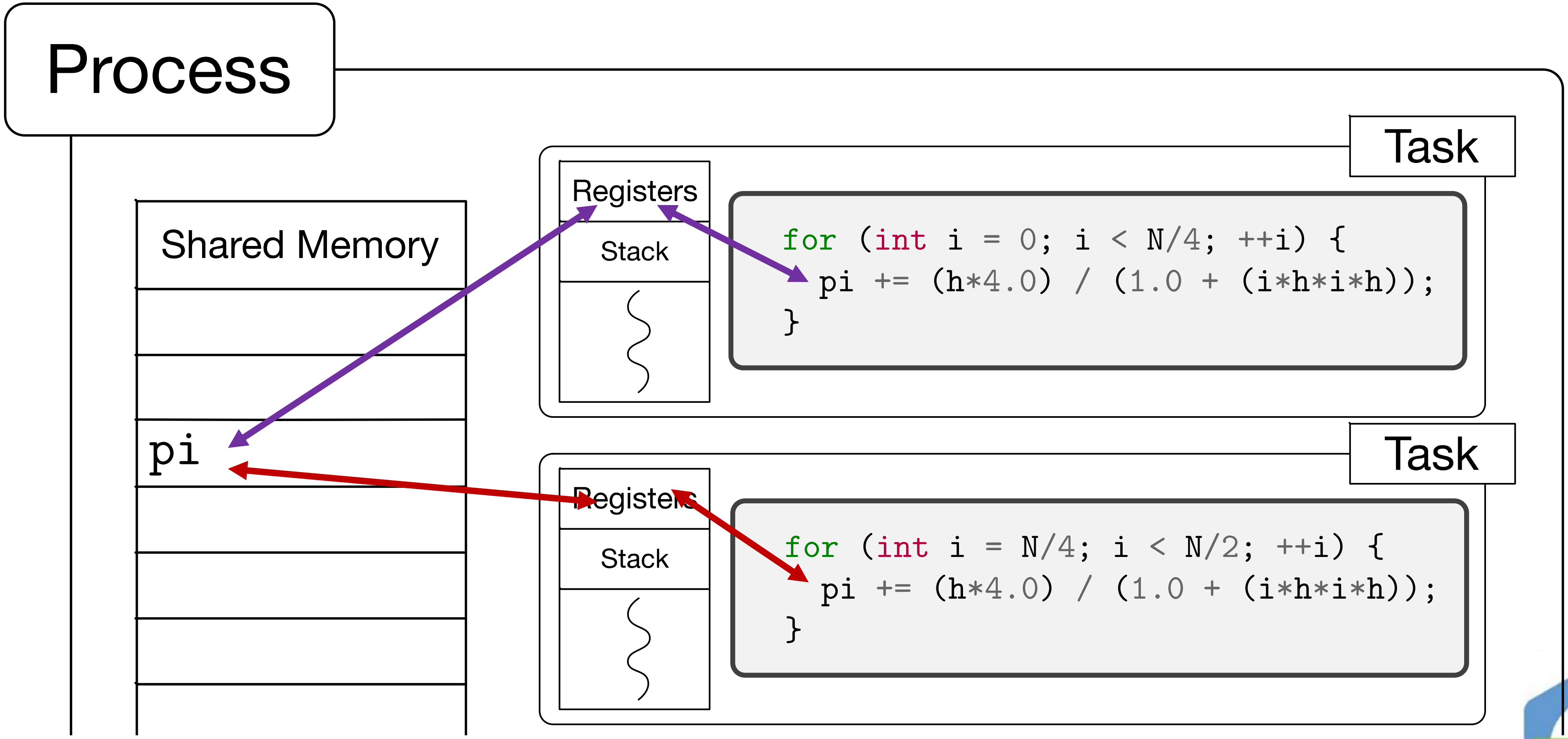


Threads

```
void pi_helper(int begin, int end, double h,
    for (int i = begin; i != end; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));
}

int main() {
    int const N = 1'000'000;
    double h = 1.0 / N;

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

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

    std::println("pi: {}", pi);
}
```



Race Condition



Synchronization, Mutual Exclusion

- `std::mutex`
  - Lock and unlock
  - Prone to deadlock
- RAII -> `std::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

Mutual exclusion region

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

void pi_helper(int begin, int end, double h) {
    for (int i = begin; i != end; ++i) {
        mtx.lock();
        pi += h * 4.0 / (1 + sqr(i * h));
        mtx.unlock();
    }
}
```

Mutex

- Locking and unlocking at every trip in inner loop

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

void pi_helper(int begin, int end, double h) {
    for (int i = begin; i != end; ++i) {
        mtx.lock();
        pi += h * 4.0 / (1 + sqr(i * h));
        mtx.unlock();
    }
}
```

- Without mutex
  - Fast, but wrong: `pi: 0.7194140023700201, time: 53 [ms]`
- With mutex
  - Slow, but correct: `pi: 3.141592663590465, time: 3422 [ms]`

Mutex

- Locking and unlocking at every function call

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

void pi_helper(int begin, int end, double h) {
    mtx.lock();
    for (int i = begin; i != end; ++i) {
        pi += h * 4.0 / (1 + sqr(i * h));
    }
    mtx.unlock();
}
```

- Without mutex
  - Fast, but wrong: `pi: 0.7194140023700201`, time: `53 [ms]`
- With mutex
  - Correct, but faster: pi: `3.141592663588645`, time: `143 [ms]`

Mutex

- Locking and unlocking at every function call

```
double pi = 0.0;
std::mutex mtx;
int const N = 1'000'000'000;     // really large number

void pi_helper(int begin, int end, double h) {
    mtx.lock();
    for (int i = begin; i != end; ++i) {
        pi += h * 4.0 / (1 + sqr(i * h));
    }
    mtx.unlock();
}
```

- With mutex:
  - Not Correct! `pi: 12.221553216452001, time: 5953 [ms]`

Aside: Integers

<table>
<tr><th rowspan="2">Equivalent type</th><th colspan="5">Width in bits by data model</th></tr>
<tr><th>C++ standard</th><th>LP32</th><th>ILP32</th><th>LLP64</th><th>LP64</th></tr>
<tr><td><code>short</code></td><td rowspan="2">at least 16</td><td rowspan="2">16</td><td rowspan="2">16</td><td rowspan="2">16</td><td rowspan="2">16</td></tr>
<tr><td><code>unsigned short</code></td></tr>
<tr><td><code>int</code></td><td rowspan="2">at least 16</td><td rowspan="2">16</td><td rowspan="2">32</td><td rowspan="2">32</td><td rowspan="2">32</td></tr>
<tr><td><code>unsigned int</code></td></tr>
<tr><td><code>long</code></td><td rowspan="2">at least 32</td><td rowspan="2">32</td><td rowspan="2">32</td><td rowspan="2">32</td><td rowspan="2">64</td></tr>
<tr><td><code>unsigned long</code></td></tr>
<tr><td><code>long long</code></td><td rowspan="2">at least 64</td><td rowspan="2">64</td><td rowspan="2">64</td><td rowspan="2">64</td><td rowspan="2">64</td></tr>
<tr><td><code>unsigned long long</code></td></tr>
</table>

Aside: Types

```
template <typename T>
void output_info() {
    std::p
        8
}

int main()
    std::p
    output
    output
    output
    output
    output
    output
    output
    output
    output
    output_info<float>();
    output_info<double>();
}
```

Type	Bytes	Bits	Min	Max
bool	1	8	false	true
short	2	16	-32768	32767
unsigned short	2	16	0	65535
int	4	32	-2147483648	2147483647
unsigned int	4	32	0	4294967295
long	4	32	-2147483648	2147483647
unsigned long	4	32	0	4294967295
int64	8	64	-9223372036854775808	9223372036854775807
unsigned int64	8	64	0	18446744073709551615
float	4	32	1.1754944e-38	3.4028235e+38
double	8	64	2.2250738585072014e-308	1.7976931348623157e+308

Mutex

- Locking and unlocking at every function call

```
double pi = 0.0;
std::mutex mtx;
int64_t const N = 1'000'000'000;      // really large number

void pi_helper(int64_t begin, int64_t end, double h) {
    mtx.lock();
    for (int64_t i = begin; i != end; ++i) {
        pi += h * 4.0 / (1 + sqr(i * h));
    }
    mtx.unlock();
}
```

- With mutex:
  - Correct and fast(er)! `pi: 3.1415926545899198, time: 1408 [ms]`

Mutex

- Locking and unlocking at every function call

```cpp
double pi = 0.0;
std::mutex mtx;
int64_t const N = 1'000'000'000;      // really large number

void pi_helper(int64_t begin, int64_t end, double h) {
    mtx.lock();
    for (int64_t i = begin; i != end; ++i) {
        pi += h * 4.0 / (1 + sqr(i * h));
    }
    mtx.unlock();
}
```

Mutual exclusion region

- With mutex (4 threads):
  - Correct and fast! `pi: 3.1415926545899198, time: 1408 [ms]`
- With mutex (1 thread):
  - Correct and fast, but does not scale! `pi: 3.1415926545899198, time: 1683 [ms]`

Back Where We Started

- What happened?
- We found concurrency (partitioned the integration)
- We had a race because shared (global) variable `pi`
- Protected each update
- Too slow
- Protected each helper
- No longer concurrent

```cpp
int main()
{
    int const N = 1'000'000;
    double pi = 0;

    for (int i = 0; i != N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/4; i != N/2; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = N/2; i != 3*N/4; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    for (int i = 3*N/4; i != N; ++i)
        pi += h * 4.0 / (1 + sqr(i * h));

    std::println("pi: {}", pi);
}
```

Mutex - done right

- Locking and unlocking at every function call

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

void pi_helper(int64_t begin, int64_t end, double h)
{
    double local_pi = 0.0;
    for (int64_t i = begin; i != end; ++i)
        local_pi += h * 4.0 / (1 + sqr(i * h));

    mtx.lock();
    pi += local_pi;
    mtx.unlock();
}
```

Mutual exclusion region

- Very short mutual exclusion region

Control Number of Threads

- Make number of threads configurable:

```
std::vector<std::thread> threads;
int64_t block_size = N / num_blocks;

for (int n = 0; n != num_blocks; ++n) {
    threads.push_back(
        std::thread(pi_helper, n * block_size, (n + 1) * block_size, h));
}

for (int n = 0; n != num_blocks; ++n) {
    threads[n].join();
}
```

Results

```
pi_threads 1000000000 1
pi: 3.1415926545900716, time: 1147 [ms]

pi_threads 1000000000 2
pi: 3.1415926545897657, time: 642 [ms]

pi_threads 1000000000 4
pi: 3.1415926545898416, time: 416 [ms]

pi_threads 1000000000 6
pi: 3.141592646589822, time: 340 [ms]

pi_threads 1000000000 8
pi: 3.141592654589780 8, time: 302 [ms]
```

- Sequential
- 2 threads, speedup of ~2
- 4 threads, speedup of ~3
- 6 threads, speedup of ~4
- 8 threads, speedup of ~4

Aside: Safe Locking

- What we ended up with:

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

void pi_helper(long long begin, long long end, double h) {
    double local_pi = 0.0;
    for (long long i = begin; i != end; ++i)
        local_pi += h * 4.0 / (1 + sqr(i * h));

    mtx.lock();
    pi += local_pi;
    mtx.unlock();
}
```

- But what if we returned without calling `unlock()`?
  - We wouldn't be able to acquire the `mutex` anymore – causing a deadlock

Aside: Safe Locking

- C++ allows to guarantee unlocking by introducing types like `std::lock_guard`:

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

void pi_helper(long long begin, long long end, double h) {
    double local_pi = 0.0;
    for (long long i = begin; i != end; ++i)
        local_pi += h * 4.0 / (1 + sqr(i * h));

    std::lock_guard l(mtx);
    pi += local_pi;
}
```

  - This type calls `lock()` in it's constructor and calls `unlock()` in it's destructor
    - Destructor is always called when the `lock_guard` goes out of scope

BTW: Our Example

- Protect access to shared variable and printing

```
std::mutex mtx;

void thread_func(int num_thread)
{
    std::lock_guard l(mtx);
    ++shared_int;
    std::println("Thread {}, stack {}, shared: {} ({})",
        num_thread, static_cast<void*>(&num_thread),
        static_cast<void*>(&shared_int), shared_int);
}
```

Critical section

Parallelize Loops

Parallelize Loops

Sequence of elements:

0	1	2	3	…	N-1	N

```
std::vector<int> d = {...};
std::for_each(d.begin(), d.end(), [](int val) {...});
```

```
template <typename Iterator, typename F>
void for_each(Iterator b, Iterator e, F f)
{
    while (b != e)
        f(*b++);
}
```

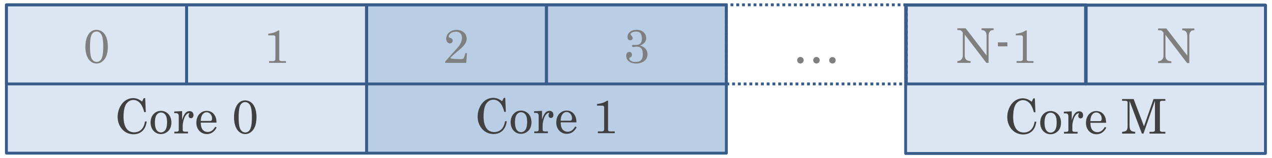
Execution Policies

- Standard introduces: `std::execution::seq`, `std:: execution::par`, `std:: execution::unseq` (C++20), `std:: execution::par_unseq`
  - Passed as additional first argument to algorithm
- Convey guarantees/requirements imposed by loop body
  - `seq`: execute in-order (sequenced) on current thread
  - `unseq`: allow out-of-order (unsequenced) execution on current thread - vectorization
  - `par`: allow parallel execution on different threads
  - `par_unseq`: allow parallel out-of-order (vectorized) execution on different threads
- Proposed for standardization (P0350: Integrating simd with parallel algorithms): `std::execution::simd`
  - Enable explicit vectorization that relies on special C++ types representing vector registers (`std::experimental::simd`, see: Parallelism TS V2, latest draft: N5001)
- HPX introduces:
  - Asynchronous policies, e.g. `par(task)`: allow asynchronous operation
  - Explicit parallelized vectorization: `par_simd`
  - Executors: attached to execution policies using `.on()`

See: wg21.link/p0350, wg21.link/n5001

Parallelize Loops

Sequence of elements:




```
std::vector<int> d = {...};
std::for_each(std::execution::par, d.begin(), d.end(), [](int val) {...});
```

```
template <typename Iterator, typename F>
void for_each(std::execution::parallel_policy, Iterator b, Iterator e, F f) {
    auto size = std::distance(b, e);                                   // Iterator should be random access
    std::vector<std::jthread> v;
    for (size_t chunk = 0; chunk != NUM_CHUNKS; ++chunk) {             // assume: cleanly divisible
        v.push_back(std::jthread([&]() {                               // launch new thread
            auto begin = std::next(b, (chunk * size) / NUM_CHUNKS);
            std::for_each(begin, std::next(begin, size / NUM_CHUNKS), f);   // sequential for_each()
        }));
    }
}      // join threads implicitly
```

Parallelize Loops: Observations

- Parallelization concurrently runs sequential operations on parts of the input
  - At least for CPU based implementations
  - GPU based algorithms are usually different
- Iterators should be random access
  - Otherwise performance might be bad
- NUM_CHUNKS is a magic number!
  - How should we select it?
  - What are the criteria for best performance?
- NUM_CORES is another magic number
- AFFINITIES are important too (NUMA awareness!), control task placement

Inter-thread Communication

Condition Variables

- Synchronization primitive that is used to notify the other threads in a multithreading environment that the shared resource is free to access
  - Defined in `#include <condition_variable>`
- Used in cases where one thread has to wait for another thread execution to continue the work
  - For example, the producer-consumer relationship, sender-receiver relationship, etc.
- The condition variable makes the calling thread wait until it is notified by the other thread
  - It is used with a `mutex` to block access to the shared resource when one thread is working on it

Condition Variables

`wait(lock, pred)`

- Causes the current thread to block until the condition variable is notified and as long as the given predicate function returns false

`notify_one()`

- If any threads are waiting on this `condition_variable`, calling `notify_one` unblocks one of the waiting threads

`notify_all()`

- Unblocks all threads currently waiting for this `condition_variable`

Condition Variables

```
// mutex to block threads
std::mutex mtx;
std::condition_variable cv;

bool data_ready = false;
int data = 0;

void producer();
void consumer();

int main() {
    std::thread consumer_thread(consumer);
    std::thread producer_thread(producer);

    consumer_thread.join();
    producer_thread.join();

    return 0;
}
```

Condition Variables

- Consumer waits for producer to call `notify_one()` after having provided the data:

```
void consumer()
{
    // locking
    std::unique_lock lock(mtx);

    // waiting
    cv.wait(lock, []() { return data_ready; });

    std::println("Data received: {}!", data);
}
```

Condition Variables

- Producer sets data and then calls `notify_all()`/`notify_one()`:

```
void producer()
{
    // lock release
    std::lock_guard lock(mtx);

    data = 42;              // set data
    data_ready = true;      // variable to avoid spurious wakeup

    // logging notification to console
    std::println("Data sent: {}!", data);

    // notify consumer when done
    cv.notify_one();
}
```

Creating a Threadpool

Threadpool Declaration

```
class thread_pool {                           // Class that represents a simple thread pool
public:
    // Constructor to create a thread pool with given number of threads
    thread_pool(size_t num_threads = std::thread::hardware_concurrency());
    ~thread_pool();                           // Destructor to stop the thread pool

    void enqueue(std::function<void()> task); // Enqueue task for execution by the pool

private:
    std::vector<std::jthread> threads_;       // Vector to store worker threads
    std::queue<std::function<void()>> tasks_; // Queue of tasks

    // Condition variable to signal changes in the state of the task queue
    std::condition_variable cv_;
    std::mutex queue_mutex_;                  // Mutex to synchronize access to shared data
    bool stop_ = false;      // Flag to indicate whether the thread pool should stop or not
};
```

Threadpool Definition

```
// Constructor to create a thread pool with given number of threads
thread_pool(size_t num_threads) {

    // Function that is executed by each of the threads in the pool
    auto thread_func = [this]() {
        while (true) {
            // pull tasks from queue and execute them
            // ...
        }
    };

    // Create worker threads
    for (size_t i = 0; i < num_threads; ++i)
        threads_.push_back(std::jthread(thread_func));
}
```

Threadpool Definition

```cpp
// Function that is executed by each of the threads in the pool
auto thread_func = [this]() {
    while (true) {
        std::function<void()> task;
        {
            // Locking the queue so that data can be shared safely
            std::unique_lock lock(queue_mutex_);

            // Wait until there is a task to execute or the pool is stopped
            cv_.wait(lock, [this] { return !tasks_.empty() || stop_; });

            // exit the thread in case the pool is stopped and there are no tasks
            if (stop_ && tasks_.empty()) return;

            task = tasks_.front(); tasks_.pop();   // Get the next task from the queue
        }
        task();    // execute the task
    }
};
```

Threadpool Definition

```
// Enqueue task for execution by the thread pool
void enqueue(std::function<void()> task)
{
    {
        std::unique_lock lock(queue_mutex_);
        tasks_.push_back(task);
    }
    cv_.notify_one();
}
```

Threadpool Definition

```
// Destructor to stop the thread pool
~thread_pool()
{
    {
        // Lock the queue to update the stop flag safely
        std::unique_lock lock(queue_mutex_);
        stop_ = true;
    }

    // Notify all threads
    cv_.notify_all();
}
```

Threadpool Example

```cpp
int main() {
    // Create a thread pool with 4 (OS-) threads
    thread_pool pool(4);

    // Enqueue tasks for execution
    for (int i = 0; i < 50; ++i) {
        pool.enqueue([i]() {
            std::println("Task {} is running on thread {}",
               i, std::this_thread::get_id());
            // Simulate some work
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        });
    }
    return 0;
}
```

Parallelize Loops (Threadpool)

Sequence of elements:

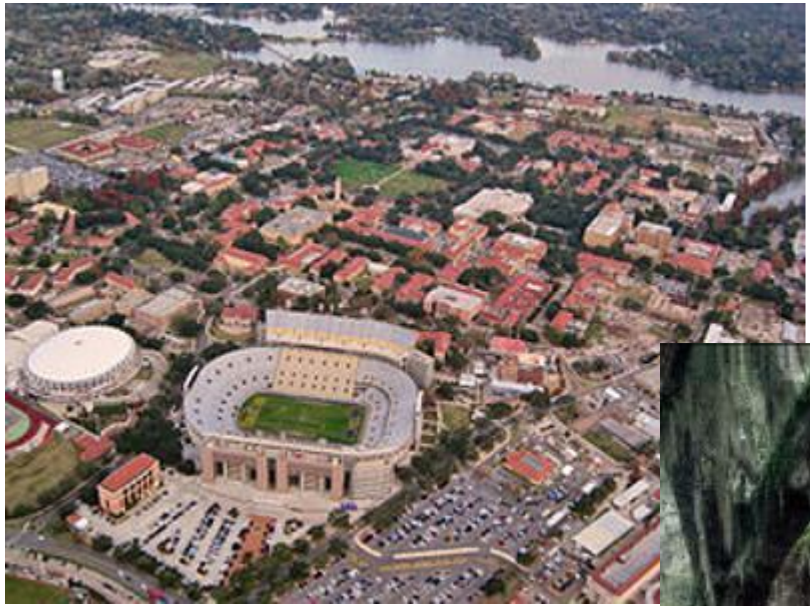
0	1	2	3	…	N-1	N
Core 0		Core 1			Core M	

```
std::vector<int> d = {...};
for_each(thread_pool(4), d.begin(), d.end(), [](int val) {...});
```

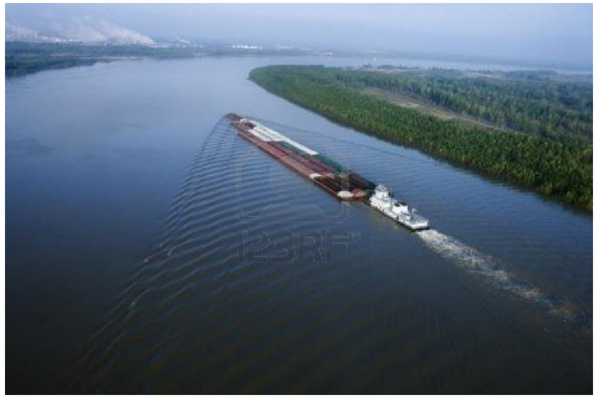
```
template <typename Iterator, typename F>
void for_each(thread_pool pool, Iterator b, Iterator e, F f) {
    auto size = std::distance(b, e);                                    // Iterator should be random access
    for (size_t chunk = 0; chunk != NUM_CHUNKS; ++chunk) {              // assume: cleanly divisible
        pool.enqueue([&]() {                                            // launch tasks
            auto begin = std::next(b, (chunk * size) / NUM_CHUNKS);
            std::for_each(begin, std::next(begin, size / NUM_CHUNKS), f);   // sequential for_each()
        });
    }
    pool.wait();      // join tasks explicitly
}
```

Conclusion

- Threads are the OS unit of concurrency
  - Abstraction of a virtual CPU core
  - Can use `std::jthread`, etc., to manage threads within a process
  - They share data → need synchronization to avoid data races
- Synchronization can be achieved using different primitives
  - `std::mutex`
  - `std::condition_variable`
  - `std::counting_semaphore`, `std::binary_semaphore`
  - `...`



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP