NORTHWEST INSTITUTE for ADVANCED COMPUTING

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle for the U.S. Department of Energy

UNIVERSITY of WASHINGTON

AMATH 483/583
Introduction to High Performance Computing

Lecture 14:
OpenMP

Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA

Overview

- Quiz
- Introduction
- OpenMP programming model
- Parallel regions
- Parallel for
- Reduction
- Race conditions

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);
}
```

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)

This (straightforward)

Became this (not so straightforward)

```
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);
}
```

```
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

Partitioned vector

Rewrite algorithm

PartitionedVector

Fork/join

C++ tasks/futures

Would need to do for all

Twice as much code

Different / separate code

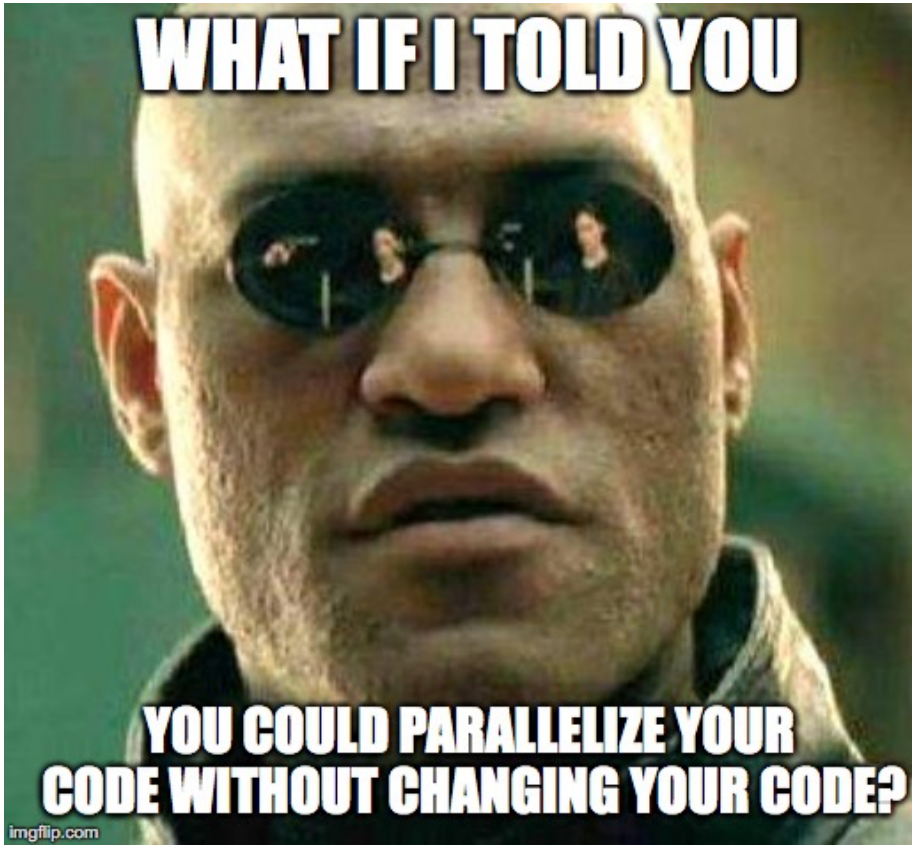
Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

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

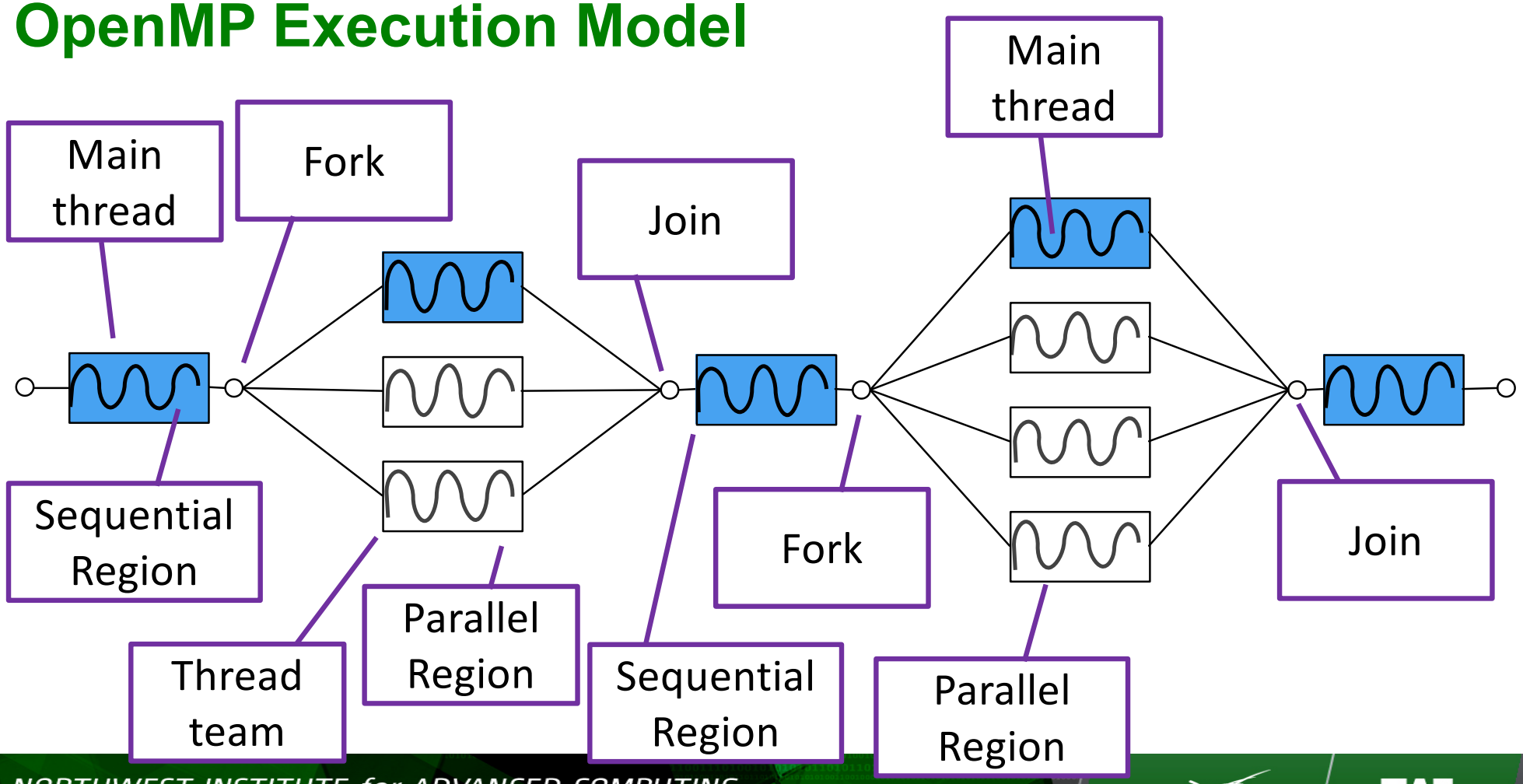
Requires no code changes

Some additions

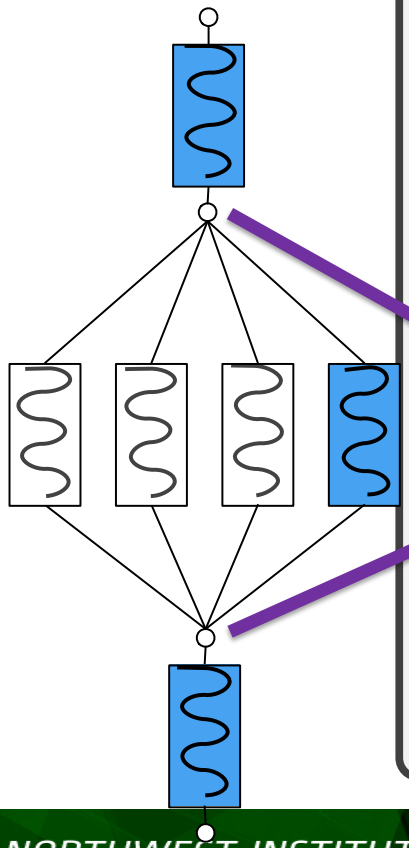
Only for parallel version

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;
}
```

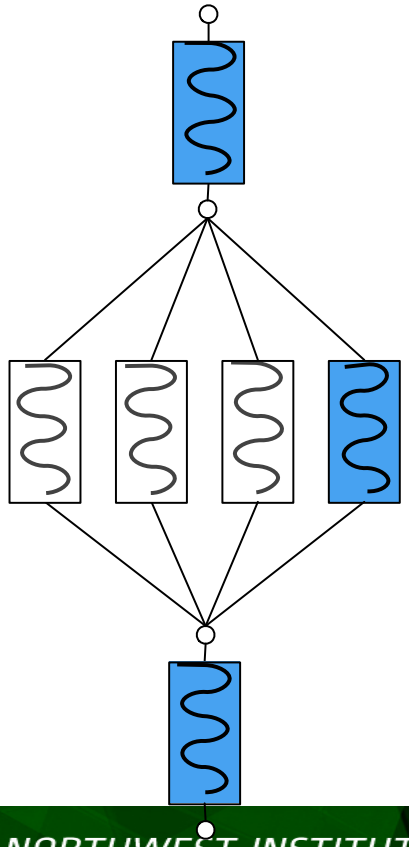
Programming with OpenMP

- 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

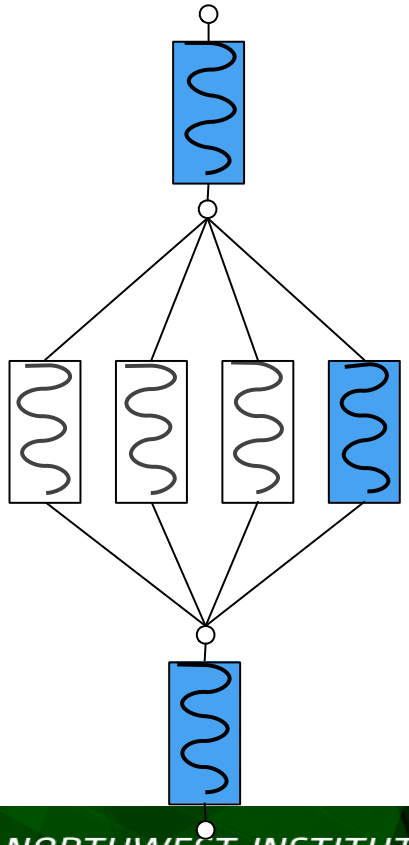


```
lums658@WE34888: ~/Teaching/amath583s19/l...
WE34888:L14/src[606] % ./pi_omp_8.exe
```

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;
}
```

Querying the environment



lums658@WE34888: ~/Teaching/amath583s19/l...

```
WE34888:L14/src[607] %
```

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 v.1

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

int main () {

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

 return 0;
}
```

Hello OpenMP

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

int main () {

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

 return 0;
}
```

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

int main () {

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

 return 0;
}
```

Explain

```
lums658@WE34888: ~/Teaching/amath583s19/l...
WE34888:L14/src[594] %
```

Hello OMP

```
#include <omp.h>

int main () {

#pragma omp parallel
 {
   size_t tid = omp_get
   std::cout << "Hello

   if (tid == 0) {
     size_t nthreads =
     std::cout <<
   }
 }

 return 0;
}
```

lums658@WE34888: ~/Teaching/amath583s19/l...

```
WE34888:L14/src[594] %
```

Comments?

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;
}
```

Output


lums658@WE34888: ~/Teaching/amath583s19/l...
WE34888:L14/src[599] %

What Happened?

Race

```
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;
}
```

Output


lums658@WE34888: ~/Teaching/amath583s19/l...
WE34888:L14/src[600] % ./pi_omp_2.exe

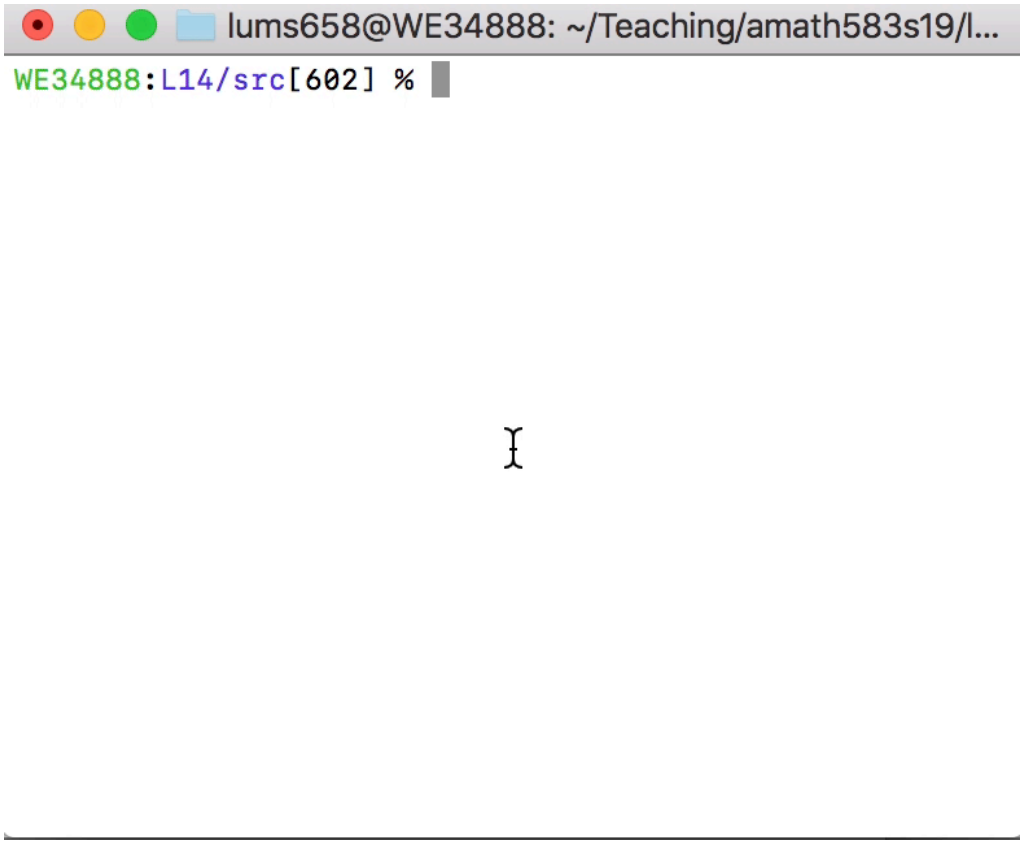
Before

```
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;
}
```

Output


lums658@WE34888: ~/Teaching/amath583s19/l...
WE34888:L14/src[602] %

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;
}
```

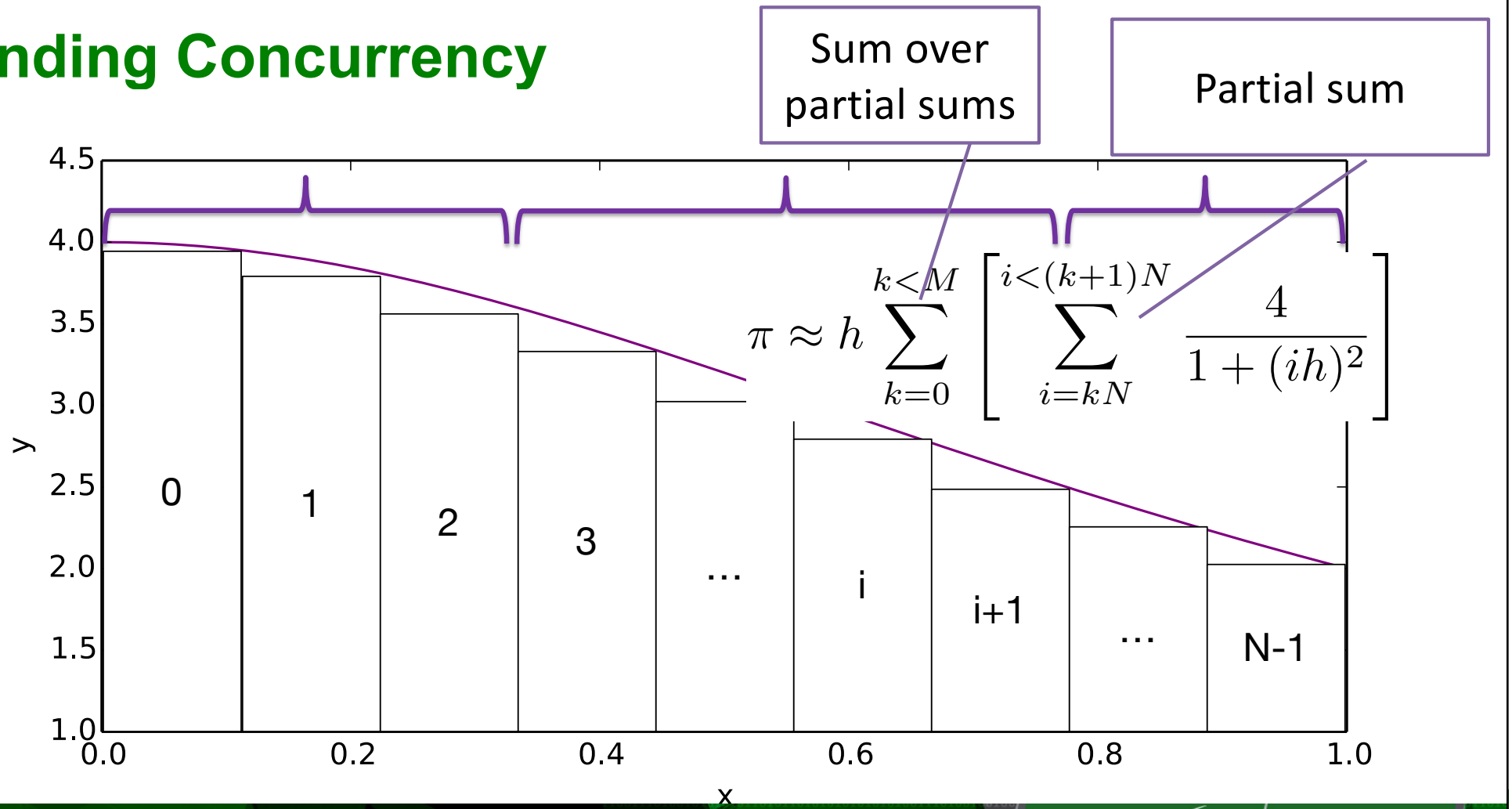
After

```
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;
}
```

Output


lums658@WE34888: ~/Teaching/amath583s19/l...
WE34888:L14/src[605] %

Finding Concurrency



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;
}
```

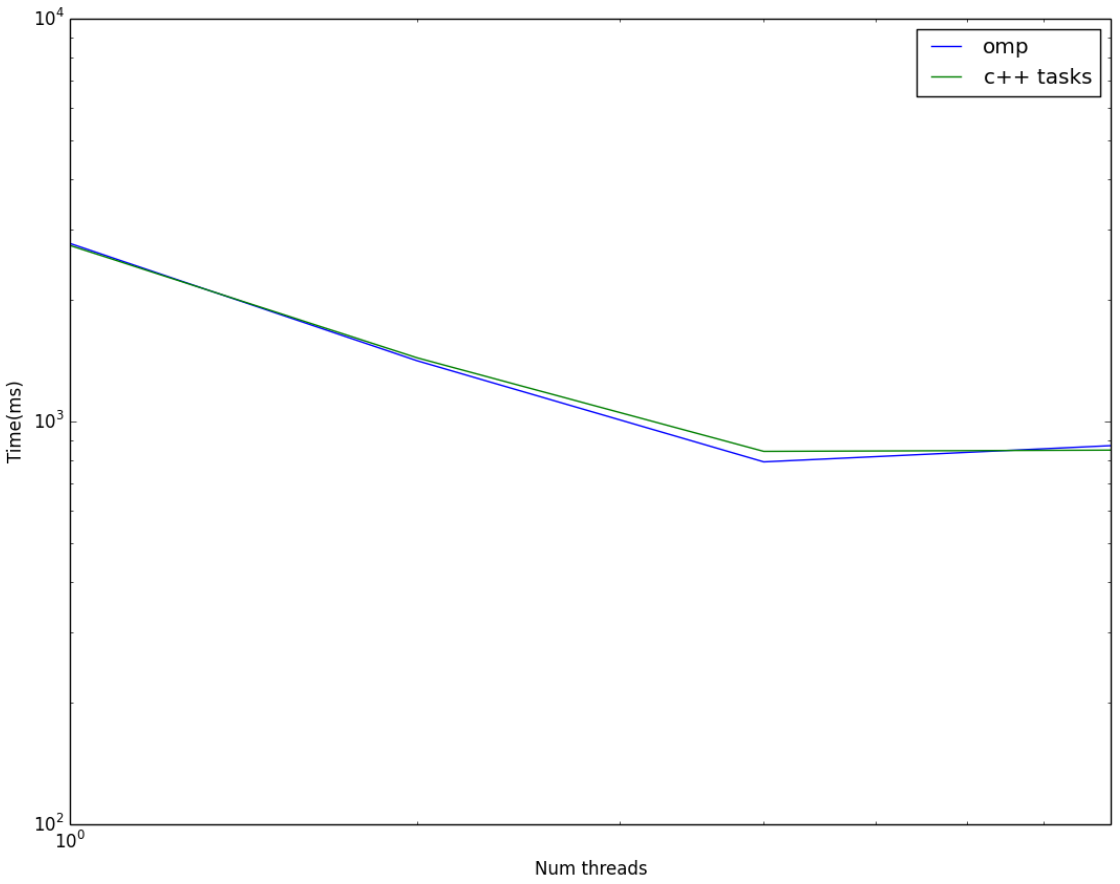
Two Norm Function (Sequential)

```
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);
}
```

Two Norm Function (Open MP)

```
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



Thank you!

Creative Commons BY-NC-SA 4.0 License