Monte Carlo Methods

Lecture 3

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

Pseudo Random Numbers

Random Numbers

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

https://xkcd.com/221/

Pseudo Random Numbers

- Deterministic, but have statistical properties resembling true random numbers
  - Common approach: each successive pseudorandom number is a function of previous
- Random pattern: Passes statistical tests (e.g., can use chi-squared)
  - Long period: Go as long as possible without repeating
  - Efficiency
  - Repeatability: Produce same sequence if started with same initial conditions
  - Portability

Example

- Use:

$$x_{n+1} = (ax_n + b)\ mod\ c$$

- Choose constants carefully, e.g.
  - a = 1664525
  - b = 1013904223
  - c = $2^{32} - 1$
- Results in integer in $[0, c)$
- Not suitable for Monte Carlo: e.g. exhibit serial correlations

Pseudo Random Numbers

- To get floating-point numbers in $[0..1)$, divide integer numbers by $c + 1$
- To get integers in range $[u.v]$, divide by $(c + 1)/(v - u + 1)$, truncate, and add u
  - Better statistics than using $mod\ (v - u + 1)$
  - Only works if u and v small compared to c
- C++:
  - Separates generation from distribution
  - `std::uniform_int_distribution<int>`
  - `std::uniform_real_distribution<double>`
  - `std::normal_distribution<double>`

https://en.cppreference.com/w/cpp/named_req/RandomNumberDistribution

Random Numbers (Don't do this)

```cpp
#include <cstdlib>    // Include for using rand
#include <ctime>      // Include for getting the current time
#include <print>

int main()
{
    // Use the current time as random seed
    std::srand(std::time(0));

    // Get one random number
    int random_value = std::rand();

    // print result
    std::println("Random value in range [0, {}]: {}", RAND_MAX, random_value);
}
```

Statistically distributed Random Numbers

```cpp
#include <print>
#include <random>     // Include for advanced random numbers

int main()
{
    // Create a random number device
    std::random_device rd;

    // Use the standard mersenne_twister_engine
    std::mt19937 gen(rd());

    // Specify the interval [1, 6]
    std::uniform_int_distribution<int> dis(1, 6);

    // Specify the interval [1.0, 6.0]
    std::uniform_real_distribution<double> disd(1, 6);

    std::println("random int: {}, real: {}", dis(gen), disd(gen));
}
```

Random Number Distribution

- Histogram allows to approximate probability distribution:

```
template <typename T>
class histogram {
    std::vector<size_t> bucket_data;
    T min_value, max_value;

public:
    histogram(size_t num_buckets, T smallest, T largest);
    void add_value(T value);

    std::vector<T> buckets() const;
    std::vector<size_t> normalized_data() const;
};
```

This is a semi-regular type!

Random Number Distribution

```
// Store num_buckets counters evenly distributed in between the smallest
// and largest values. It will also store counters in two additional
// buckets, one for values smaller and one for values larger than the range
// given during construction.

histogram(size_t num_buckets, T smallest, T largest)
  : bucket_data(num_buckets + 2)
  , min_value(smallest)
  , max_value(largest)
{}
```

Random Number Distribution

```
// Increment the counter for the appropriate bucket, which index is computed
// based on the number of buckets and the smallest and largest values that
// were used during construction.

void add_value(T value) {
    // for out-of-range values smaller than minimum use first bucket
    size_t index = 0;
    if (value > max_value) {
        // for out-of-range values larger than maximum use last bucket
        index = bucket_data.size() - 1;
    } else if (value >= min_value) {
        auto bucket_size = (max_value - min_value) / (bucket_data.size() - 2);
        index = static_cast<size_t>((value - min_value) / bucket_size) + 1;
    }
    ++bucket_data[index];
}
```

Random Number Distribution

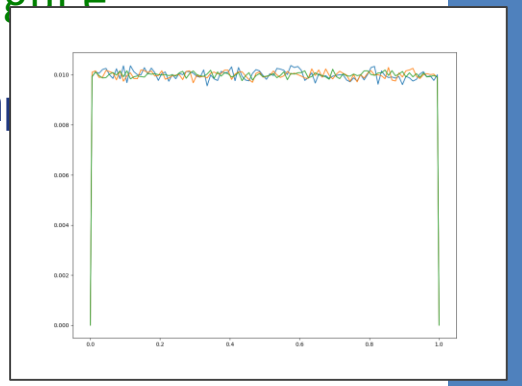
```
// Return a vector of mid-point values for each bucket.
auto buckets() const {
    std::vector<T> buckets;
    buckets.reserve(bucket_data.size() - 2);
    buckets.push_back(min_value);

    T delta = (max_value - min_value) / (bucket_data.size() - 2);
    for (size_t i = 0; i != bucket_data.size() - 2; ++i)
        buckets.push_back(min_value + i * delta + 0.5 * delta);

    buckets.push_back(max_value);
    return buckets;
}
```

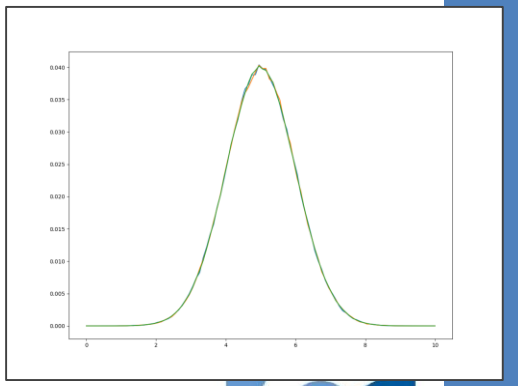
Random Number Distribution

```
// Return a vector of normalized counts for each bucket collected.

auto normalized_data() const {
    std::vector<double> data;
    data.reserve(bucket_data.size());

    auto max = std::max_element(bucket_data.begin(), bucket_data.end());
    std::transform(
        bucket_data.begin(), bucket_data.end(),
        std::back_inserter(data),
        [&](size_t value) { return double(value) / *max; });

    return data;
}
```

Plotting Distributions

```
plt::figure_size(1280, 960); // Initialize a 1280 X 960 figure

for (int max_attempts = 17; max_attempts < 21; ++max_attem
{
    int const n_total = 1 << max_attempts;  // 2^N

    histogram<double> hist(100, 0.0, 1.0);
    for (int n = 0; n < n_total; ++n)
        hist.add_value(get_random_number());

    plt::plot(hist.buckets(), hist.normalized_data());
}

plt::save("random_numbers.png");
```



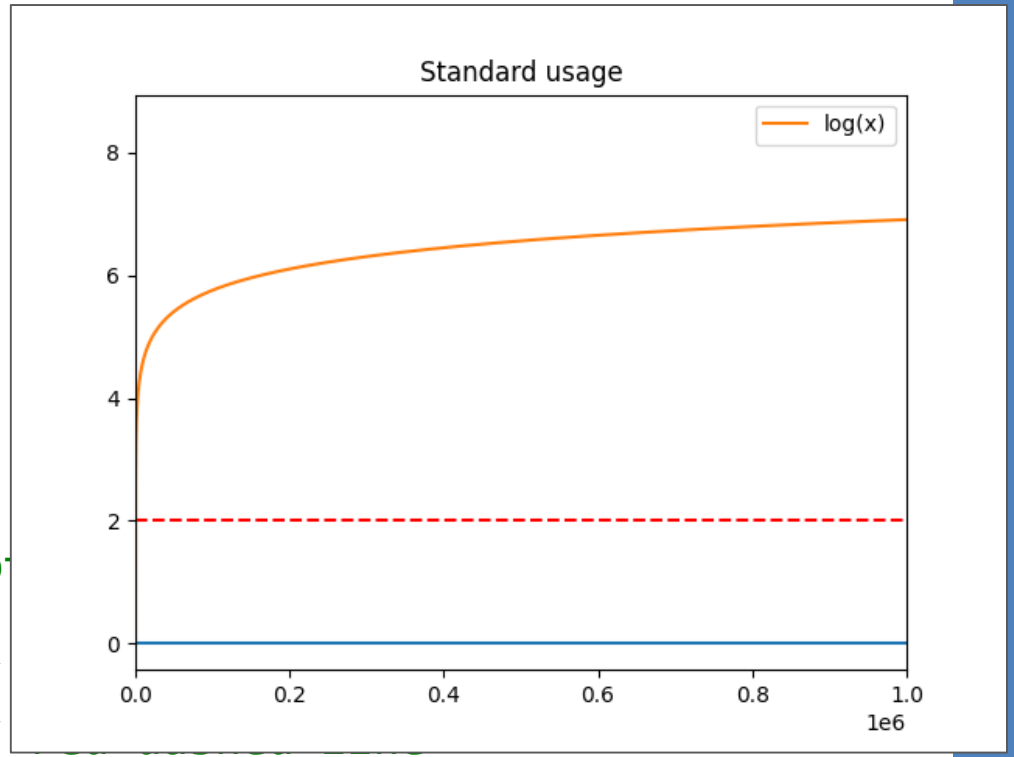


Aside: Generating Graphs

- We will use `matplotlib.h`, a wrapper for Pythons matplotlib module
- Documentation can be found at https://matplotlib-cpp.readthedocs.io/
- Pre-installed on Docker image used for assignments
- Minimal example:

```cpp
#include "matplotlibcpp.h"
#include <vector>

namespace plt = matplotlibcpp;

int main() {
    std::vector<double> y = {1, 3, 2, 4};
    plt::plot(y);
    plt::save("minimal.png");
}
```

```cpp
int n = 5000; // 5000 data points
std::vector<double> x(n), y(n), z(n), w(n,
for (int i = 0; i < n; ++i) {
    x[i] = i * i;
    y[i] = std::sin(2 * pi * i / 360.0);
    z[i] = std::log(i);
}

plt::figure(); // declare a new figure (op

plt::plot(x, y);                          //
plt::plot(x, w, "r--");                   //
plt::plot(x, z, {{"label", "log(x)"}}); // legend label "log(x)"

plt::xlim(0, 1000 * 1000);      // x-axis interval: [0, 1e6]
plt::title("Standard usage");   // set a title
plt::legend();                  // enable the legend

plt::save("standard.png");      // save the figure
```



Monte-Carlo Simulations

Monte Carlo Simulation

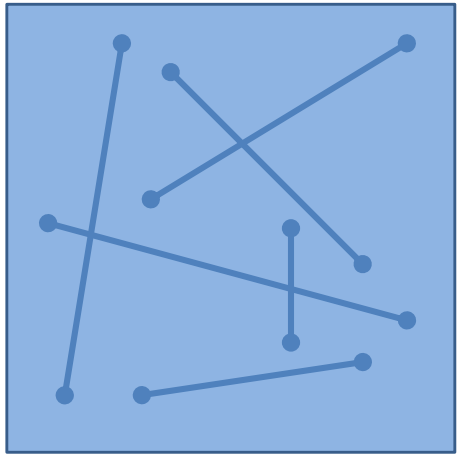
- Used when it is not feasible or impossible to compute an exact result with a deterministic algorithm
- Especially useful in
  - Studying systems with a large number of coupled degrees of freedom
    - Fluids, disordered materials, strongly coupled solids, cellular structures
  - For modeling phenomena with significant uncertainty in inputs
    - The calculation of risk in business
  - These methods are also widely used in mathematics
    - The evaluation of definite integrals, particularly multidimensional integrals with complicated boundary conditions

Monte Carlo Methods

- Monte Carlo methods are computational algorithms
  - Rely on repeated random sampling to obtain numerical results
- Three problem classes:
  - 1. Probability distributions
  - 2. Optimization
  - 3. Numerical integration
- General pattern:
  - 1. Define the input parameters
  - 2. Randomly chose input parameters
  - 3. Do deterministic computations on the inputs
  - 4. Aggregate the results

Example: Line Length

- Suppose you have a 1×1 square
  - If two points are randomly picked within the square, what is the expected value (average) of the distance between them?
- Exact answer: $I = \frac{2+\sqrt{2}+5*\text{arcsinh}(1)}{15} \approx 0.52140543316472...$
- Algorithm
  - while $n < n_{total}$ repeat
    - Get two random number pairs (x, y)
      - where: x, y: $[0, 1]$
    - Calculate length
      - $length = \sqrt{(x_2 - x_1)^2+(y_2 - y_1)^2}$
    - Accumulate overall length, update n
  - Compute average length



$l = 1$

Interactive question:

Which features of C++ do we need?

Loop, random numbers, `std::print`

Random Numbers

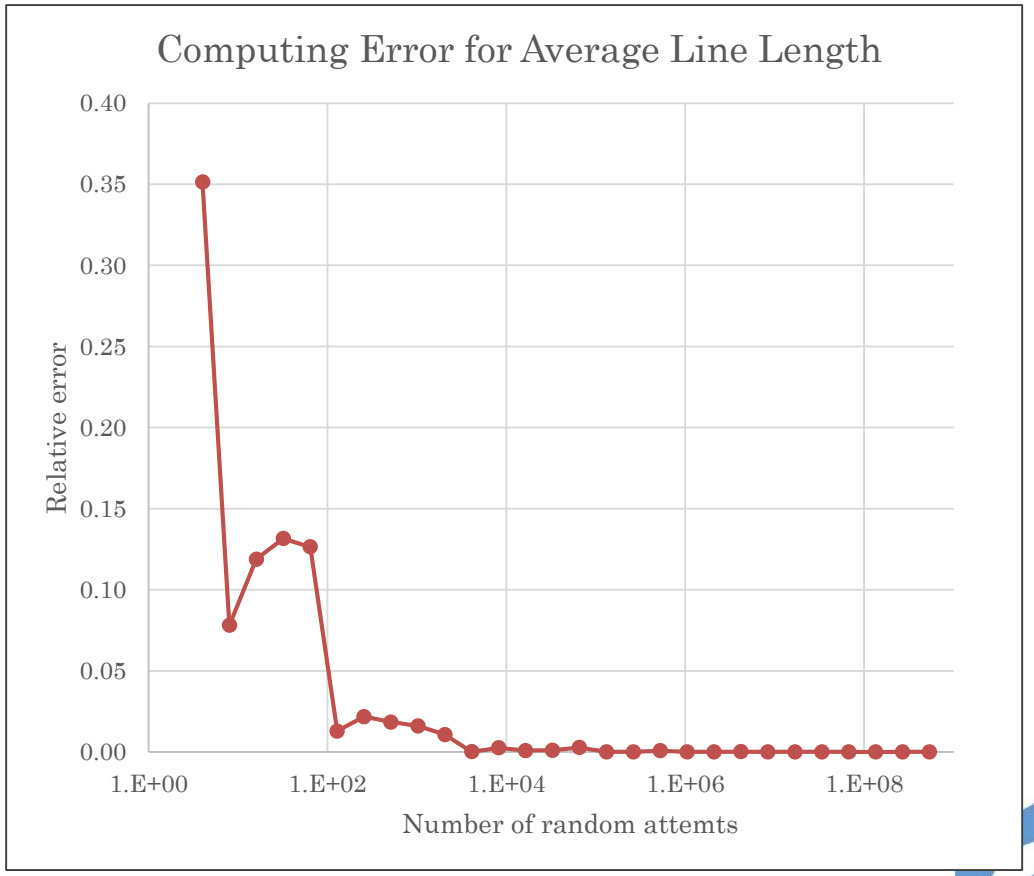
```cpp
#include <print>
#include <random>

// Initialize random number generator
std::random_device rd;
std::mt19937 gen(rd());

double get_random_number()
{
    std::uniform_real_distribution<double> dis(0.0, 1.0);
    return dis(gen);
}


// Helper function to square a number
double sqr(double x)
{
    return x * x;
}
```

Average Line Length

```
int main()
{
    std::println("attempts,average length");

    int const n_total = 10000;
    double accumulated_length = 0.0;

    for (int n = 0; n < n_total; ++n)
    {
        double x1 = get_random_number();
        double x2 = get_random_number();
        double y1 = get_random_number();
        double y2 = get_random_number();
        accumulated_length += std::sqrt(sqr(x2 - x1) + sqr(y2 - y1));
    }

    double length = accumulated_length / n_total;
    std::println("{},{}", n_total, length);

    return 0;
}
```

How many Attempts are needed?

attempts	average length	error
2	0.674785848	2.94E-01
4	0.338191643	3.51E-01
8	0.562140325	7.81E-02
16	0.583379611	1.19E-01
32	0.589994035	1.32E-01
64	0.587350866	1.26E-01
128	0.528131378	1.29E-02
256	0.510076918	2.17E-02
512	0.531026183	1.85E-02
1024	0.513027098	1.61E-02
2048	0.515803633	1.07E-02
4096	0.521301527	1.99E-04
8192	0.522698539	2.48E-03
16384	0.521847718	8.48E-04
32768	0.52195183	1.05E-03
65536	0.522813535	2.70E-03
131072	0.521373121	6.20E-05
262144	0.521448659	8.29E-05
524288	0.521805565	7.67E-04
1048576	0.521434376	5.55E-05
2097152	0.521393597	2.27E-05
4194304	0.521495376	1.72E-04
8388608	0.521449764	8.50E-05
16777216	0.521396853	1.65E-05
33554432	0.52139758	1.51E-05
67108864	0.521436921	6.04E-05
134217728	0.521400739	9.00E-06
268435456	0.521383023	4.30E-05
536870912	0.521418749	2.55E-05



Example: Winning with Dice

- Would it be profitable given 24 rolls of a pair of fair dice to bet against there being at least one double six?
- Exact Answer: $1 - \left(\frac{35}{36}\right)^{24} \approx 0.49141 \ldots$
- Algorithm:
- while $n < n_{total}$ repeat
  - Get 24 pairs of two random integers in range [1, 6]
  - If for one of them both are equal to six: count win
- $Probability = wins \,/\, n\_total$

Random Numbers

```cpp
#include <print>
#include <random>

// Initialize random number generator
std::random_device rd;
std::mt19937 gen(rd());

int roll_dice()
{
    std::uniform_int_distribution<int> dis(1, 6);
    return dis(gen);
}
```

Roll Dice

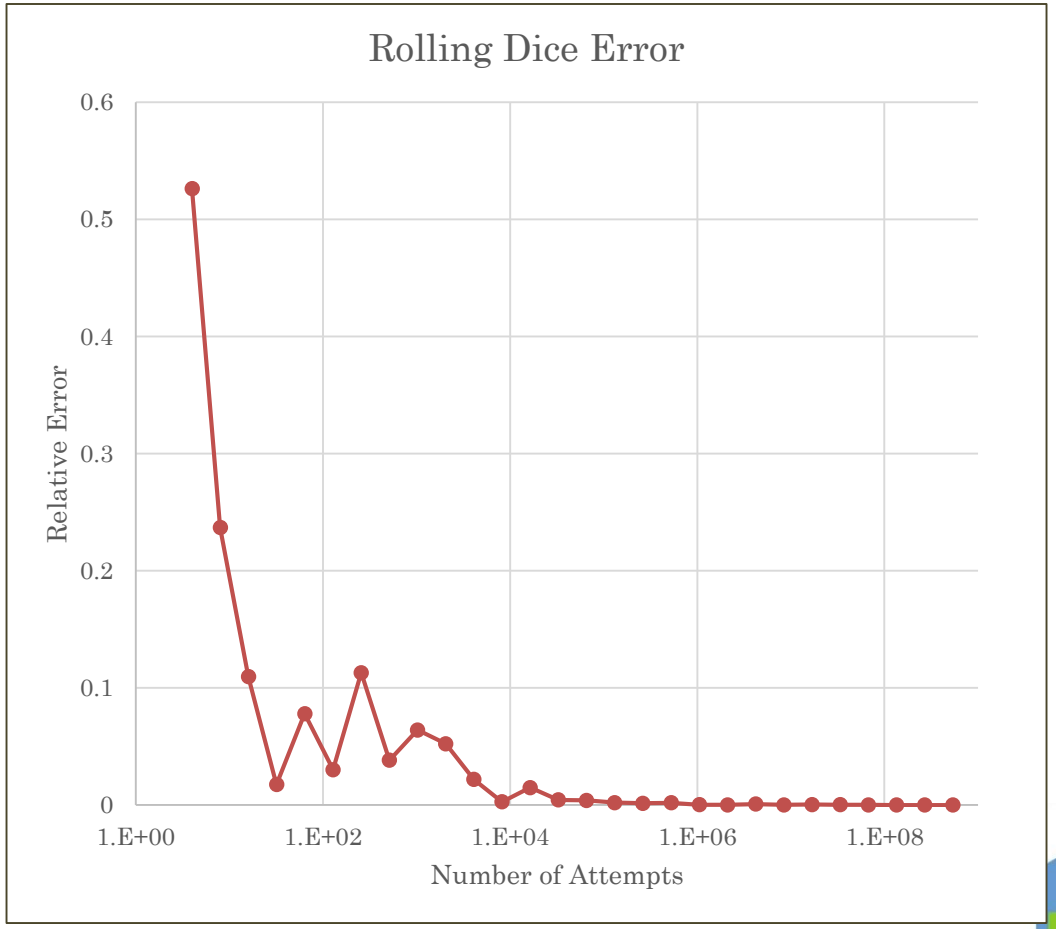
```
bool attempt_to_win()
{
    for (int r = 0; r < 24; ++r)
    {
        int r1 = roll_dice();
        int r2 = roll_dice();
        if (r1 == 6 && r2 == 6)
        {
            return true;
        }
    }
    return false;
}
```

Winning with Dice

```cpp
int main()
{
    std::println("attempts,probability");

    int const n_total = 10000;

    int num_wins = 0;
    for (int n = 0; n < n_total; ++n)
    {
        if (attempt_to_win())
        {
            ++num_wins;
        }
    }

    double probability_of_win = static_cast<double>(num_wins) / n_total;
    std::println("{},{}", n_total, probability_of_win);
}
```

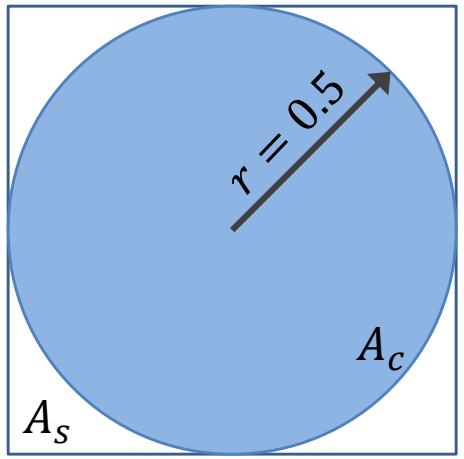
How many Attempts are needed?

attempts	probability	error
2	0.5	1.75E-02
4	0.75	5.26E-01
8	0.375	2.37E-01
16	0.4375	1.10E-01
32	0.5	1.75E-02
64	0.453125	7.79E-02
128	0.4765625	3.02E-02
256	0.546875	1.13E-01
512	0.47265625	3.82E-02
1024	0.459960938	6.40E-02
2048	0.517089844	5.23E-02
4096	0.480712891	2.18E-02
8192	0.490112305	2.63E-03
16384	0.484130859	1.48E-02
32768	0.493560791	4.39E-03
65536	0.489471436	3.93E-03
131072	0.492385864	2.00E-03
262144	0.492095947	1.41E-03
524288	0.490514755	1.81E-03
1048576	0.491295815	2.20E-04
2097152	0.491357327	9.47E-05
4194304	0.491046906	7.26E-04
8388608	0.491469502	1.34E-04
16777216	0.491607308	4.14E-04
33554432	0.491272002	2.68E-04
67108864	0.491418511	2.98E-05
134217728	0.491410904	1.43E-05
268435456	0.491398305	1.13E-05
536870912	0.491401931	3.96E-06



Example: Estimate the value of π

- Ingredients
  - Unit square $1x1$
  - Circle with the radius of $r = \frac{1}{2}$
  - Area of the circle $A_c = \pi r^2 = \frac{\pi}{4}$
  - Area of the square $A_s = 1x1 = 1$
  - Recall: $A_c = \frac{\pi}{4} \rightarrow 4A_c = \pi$

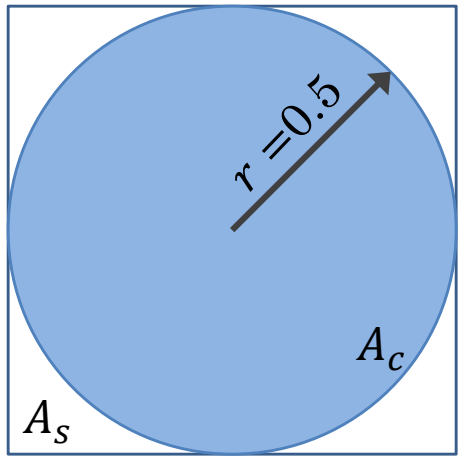


$l = 1$

Example: Estimate the value of π

- Now compute π by using the two areas:

$$\pi = 4\frac{A_c}{A_s}$$



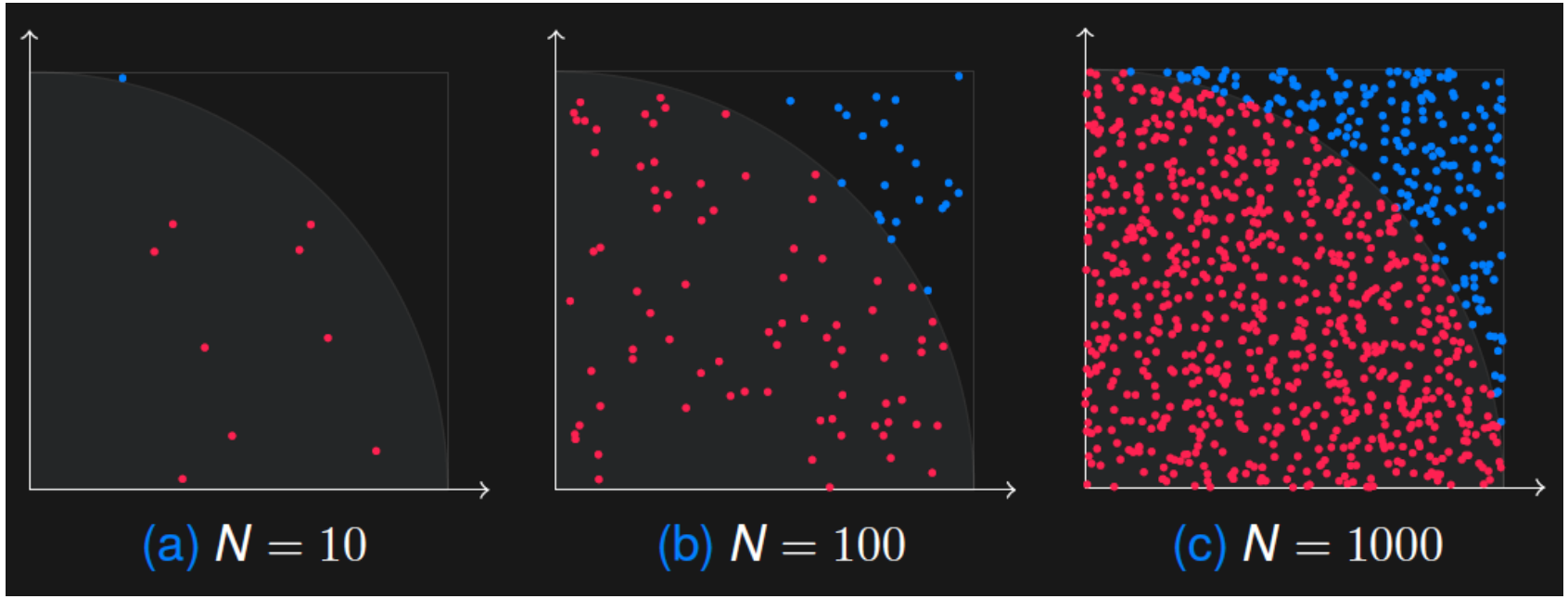
- The areas can be approximated by using N random samples (x, y) and count the points inside the circle N_c:

$$\pi \approx 4\frac{N_c}{N}$$

Algorithm

- while $n < n_{total}$ repeat
  - Generate random coordinates $(x, y) \in \mathbb{R}^2$
  - Check if $x^2 + y^2 \leq 1$
    - Increment n_c if ≤ 1
  - Increment number of attempts n
- Calculate and print $4\frac{n_c}{n_{total}}$

Example for various Numbers of Attempts



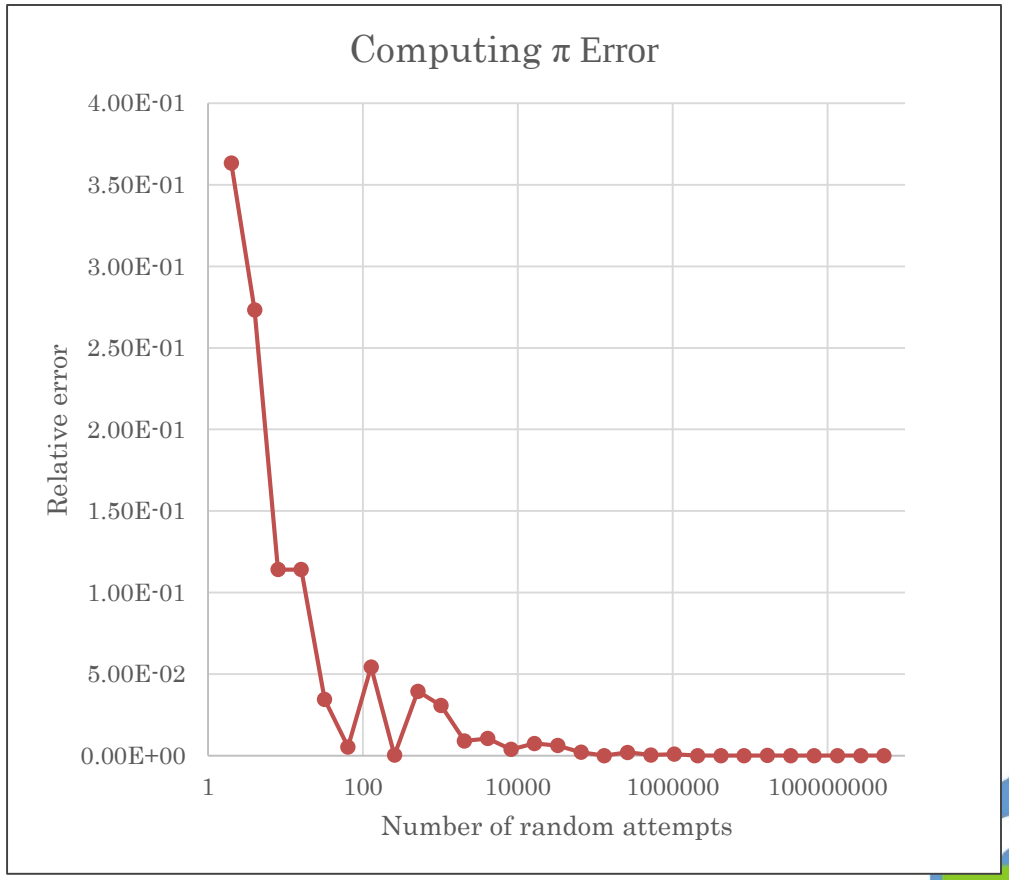
- Distribution of the point inside the circle (red) and outside of the circle (blue) for $N = 10$, $N = 100$, and $N = 1000$ random numbers.

Calculate π

```
int main()
{
    int const n_total = 10000;

    int nc = 0;
    for (int n = 0; n < n_total; ++n)
    {
        double x = get_random_number();
        double y = get_random_number();
        if (sqr(x) + sqr(y) <= 1.0)
        {
            ++nc;
        }
    }

    std::println("Approximated pi for {} attempts: {}", n_total, (4.0 * nc) / n_total);
}
```

How many random Numbers are needed?

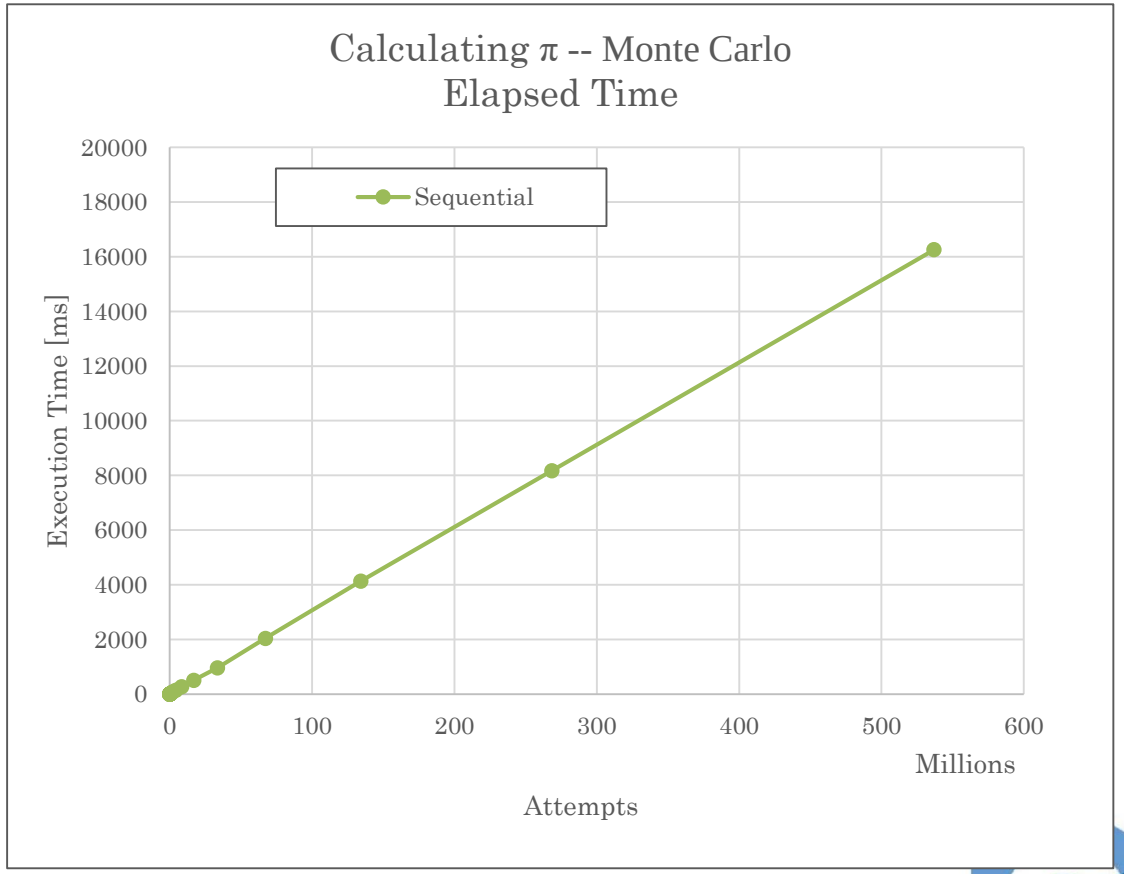
attempts	pi	error
2	2	3.63E-01
4	4	2.73E-01
8	3.5	1.14E-01
16	3.5	1.14E-01
32	3.25	3.45E-02
64	3.125	5.28E-03
128	3.3125	5.44E-02
256	3.140625	3.08E-04
512	3.265625	3.95E-02
1024	3.23828125	3.08E-02
2048	3.169921875	9.02E-03
4096	3.174804688	1.06E-02
8192	3.153808594	3.89E-03
16384	3.117919922	7.54E-03
32768	3.161254883	6.26E-03
65536	3.134887695	2.13E-03
131072	3.14151001	2.63E-05
262144	3.147781372	1.97E-03
524288	3.142829895	3.94E-04
1048576	3.138744354	9.07E-04
2097152	3.141403198	6.03E-05
4194304	3.141534805	1.84E-05
8388608	3.141631603	1.24E-05
16777216	3.141290188	9.63E-05
33554432	3.141633153	1.29E-05
67108864	3.141349852	7.73E-05
134217728	3.141705692	3.60E-05
268435456	3.141563967	9.13E-06
536870912	3.141544454	1.53E-05

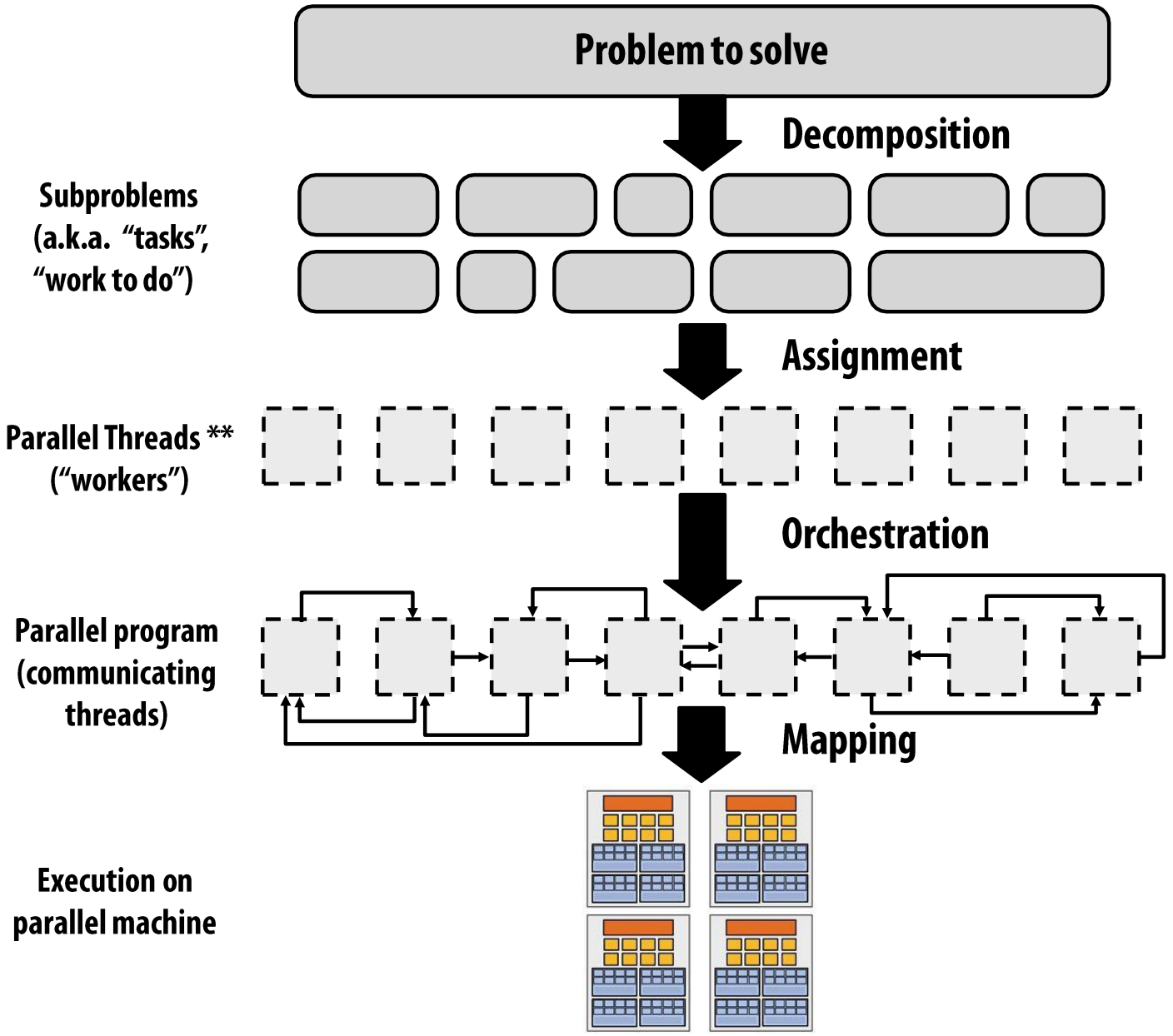


Calculate π in Parallel

Calculate π Sequentially, Timing

Attempts	pi	Error	Elapsed [ms]
2	4	0.273239545	0
4	3	0.045070341	0
8	2.5	0.204225285	0
16	3	0.045070341	0
32	3	0.045070341	0
64	2.75	0.124647813	0
128	3.28125	0.044454314	0
256	3.234375	0.029533538	0
512	3.1171875	0.007768402	0
1024	3.11328125	0.0090118	0
2048	3.236328125	0.030155237	0
4096	3.129882813	0.003727358	0
8192	3.140625	0.000308014	0
16384	3.144775391	0.001013097	0
32768	3.146484375	0.001557083	1
65536	3.129272461	0.003921639	2
131072	3.141235352	0.000113733	6
262144	3.14289856	0.000415683	10
524288	3.143165588	0.000500681	20
1048576	3.142009735	0.000132761	38
2097152	3.143692017	0.000668248	82
4194304	3.140839577	0.000239712	137
8388608	3.141868591	8.78E-05	268
16777216	3.141479254	3.61E-05	506
33554432	3.141817093	7.14E-05	959
67108864	3.141506851	2.73E-05	2040
134217728	3.141727	4.28E-05	4135
268435456	3.141671851	2.52E-05	8169
536870912	3.141656928	2.05E-05	16252




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

Calculate π in Parallel

- Computation of π in parallel is what is called 'embarrassingly' parallel
  - Because it is embarrassingly easy to do so
- Decomposition is trivial
  - Loop iterations are independent and can run in any order, or even concurrently
- Assignment to workers is equally trivial when using HPX facilities
- Orchestration trivial because there are no interdependencies
- Mapping to hardware done by HPX

Calculate π in Parallel

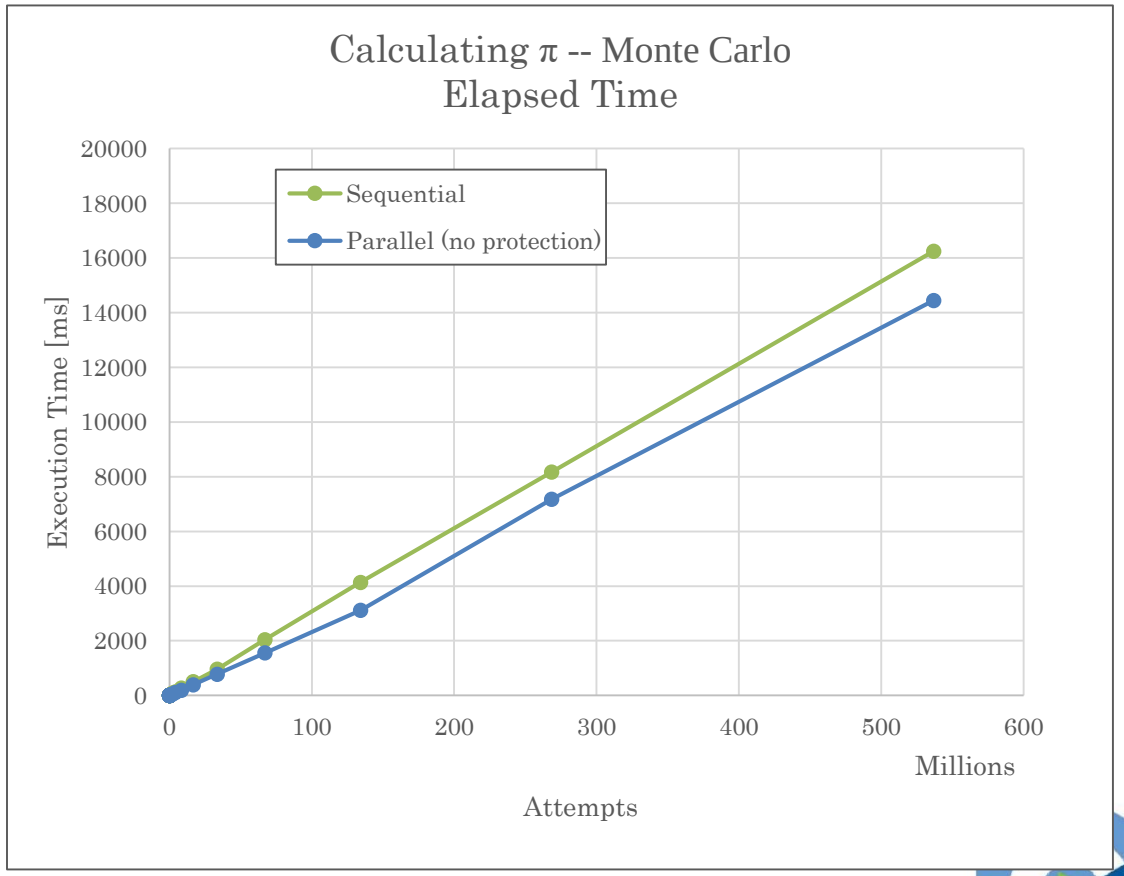
```
int main() {
    int const n_total = 10000;
    int nc = 0;

    hpx::experimental::for_loop(0, n_total,
        [&](int n) {
            double x = get_random_number();
            double y = get_random_number();
            if (sqr(x) + sqr(y) <= 1.0)
                ++nc;
        }
    );
    std::println("Approximated pi for {} attempts: {}",
        n_total, (4.0 * nc) / n_total);
}
```

Calculate π in Parallel

```
int main() {
    int const n_total = 10000;
    int nc = 0;

    hpx::experimental::for_loop(hpx::execution::par, 0, n_total,
        [&](int n) {
            double x = get_random_number();
            double y = get_random_number();
            if (sqr(x) + sqr(y) <= 1.0)
                ++nc;
        }
    );
    std::println("Approximated pi for {} attempts: {}",
        n_total, (4.0 * nc) / n_total);
}
```

Calculate π in Parallel (no protection)

Attempts	pi	Error	Elapsed [ms]
2	4	0.273239545	0
4	4	0.273239545	0
8	3	0.045070341	0
16	3	0.045070341	0
32	2.75	0.124647813	0
64	3.25	0.03450713	0
128	3.1875	0.014612762	0
256	2.25	0.283802756	0
512	1.953125	0.378301004	0
1024	1.953125	0.378301004	0
2048	1.15625	0.631954194	0
4096	1.108398438	0.64718582	0
8192	1.505859375	0.520670074	0
16384	1.155517578	0.632187331	0
32768	1.228393555	0.608990187	0
65536	1.444702148	0.540137024	1
131072	1.245727539	0.603472609	3
262144	1.131973267	0.639681718	6
524288	1.471549988	0.531591091	18
1048576	1.417877197	0.548675671	32
2097152	1.244888306	0.603739745	53
4194304	1.068852425	0.659773706	104
8388608	1.022730827	0.674454667	193
16777216	1.044085741	0.667657187	391
33554432	1.046907425	0.666759017	781
67108864	1.042102456	0.668288486	1556
134217728	1.049701184	0.665869736	3114
268435456	1.031741291	0.671586547	7176
536870912	1.033411726	0.671054831	14448



Calculate π in Parallel

```
int main() {
    int const n_total = 10000;
    int nc = 0;

    hpx::experimental::for_loop(hpx::execution::par, 0, n_total,
        [&](int n) {
            double x = get_random_number();
            double y = get_random_number();
            if (sqr(x) + sqr(y) <= 1.0)
                ++nc;
        }
    );
    std::println("Approximated pi for {} attempts: {}",
        n_total, (4.0 * nc) / n_total);
}
```

Aside: Race Conditions

- What are the possible values of `x` below after all threads finish?
- Initially `x == 0` and `y == 0`

Thread A	Thread B
`x = 1;`	`y = 2;`

- Must be **1**. Thread B does not interfere.

Aside: Race Conditions

- What are the possible values of `x` below?
- Initially `x == 0` and `y == 0`

Thread A	Thread B
`x = y + 1;`	`y = 2;`
	`y = y * 2;`

- `1` or `3` or `5` (non-deterministic)
- Race Condition: Thread A races against Thread B

Relevant Definitions

- Synchronization: Coordination among threads, usually regarding shared data
- Mutual Exclusion: Ensuring only one thread does a particular thing at a time (one thread excludes the others)
  - Type of synchronization
- Critical Section: Code exactly one thread can execute at once
  - Result of mutual exclusion
- Lock: An object only one thread can hold at a time
  - Provides mutual exclusion
  - Also called Mutex

Aside: Mutexes

- Mutexes provide two atomic operations:
  - `std::mutex::acquire()` – wait until mutex is free; then mark it as busy
    - After this returns, we say the calling thread holds the lock
  - `std::mutex::release()` – mark mutex as free
    - Should only be called by a thread that currently holds the mutex
    - After this returns, the calling thread no longer holds the mutex
    - Wake up one of the threads waiting in `acquire`
- Usage this in conjunction with locking helper types
  - `std::lock_guard`
  - `std::unique_lock`
  - …
- Locks implemented based on one of C++ main paradigms: RAII
  - Constructor acquires mutex
  - Destructor releases mutex

Aside: C++ Lock Guards

```
#include <mutex>

int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
  std::lock_guard lock(global_mutex);
  …
  ++global_i;
  // Mutex released when ‘lock’ goes out of scope
}
```

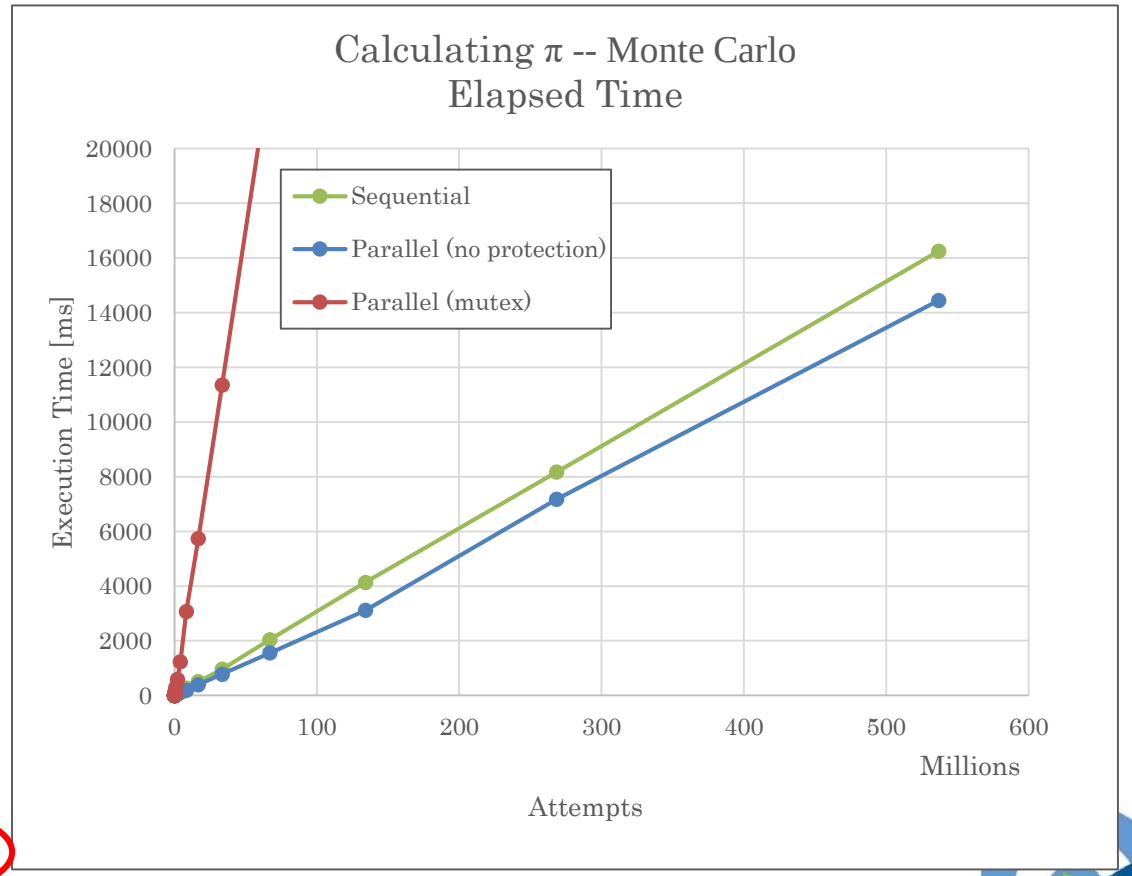
Calculate π in Parallel (mutex)

```
int main() {
    int const n_total = 10000;
    int nc = 0;

    hpx::mutex mtx;
    hpx::experimental::for_loop(hpx::execution::par, 0, n_total,
        [&](int n) {
            double x = get_random_number();
            double y = get_random_number();
            if (sqr(x) + sqr(y) <= 1.0) {
                std::unique_lock l(mtx);
                ++nc;
            }
        });
    std::println("Approximated pi for {} attempts: {}",
        n_total, (4.0 * nc) / n_total);
}
```

Critical section

Calculate π in Parallel (mutex)

Attempts	pi	Error	Elapsed [ms]
2	4	0.273239545	0
4	3	0.045070341	0
8	3.5	0.114084602	0
16	3	0.045070341	0
32	3.125	0.005281606	0
64	3.4375	0.094190234	0
128	3.1875	0.014612762	0
256	3.015625	0.040096749	0
512	3.203125	0.019586354	0
1024	3.14453125	0.000935384	0
2048	3.1484375	0.002178782	0
4096	3.103515625	0.012120295	1
8192	3.119628906	0.006991278	2
16384	3.134277344	0.002328535	4
32768	3.150512695	0.002839337	9
65536	3.144775391	0.001013097	19
131072	3.142944336	0.000430254	42
262144	3.141387939	6.52E-05	94
524288	3.143379211	0.000568679	159
1048576	3.140930176	0.000210873	293
2097152	3.140808105	0.000249729	594
4194304	3.14125061	0.000108876	1233
8388608	3.141716003	3.93E-05	3068
16777216	3.14099884	0.000189017	5742
33554432	3.141767263	5.56E-05	11356
67108864	3.141917825	0.000103505	22906
134217728	3.141463876	4.10E-05	45310
268435456	3.141514053	2.50E-05	90517
536870912	3.141645461	1.68E-05	182228

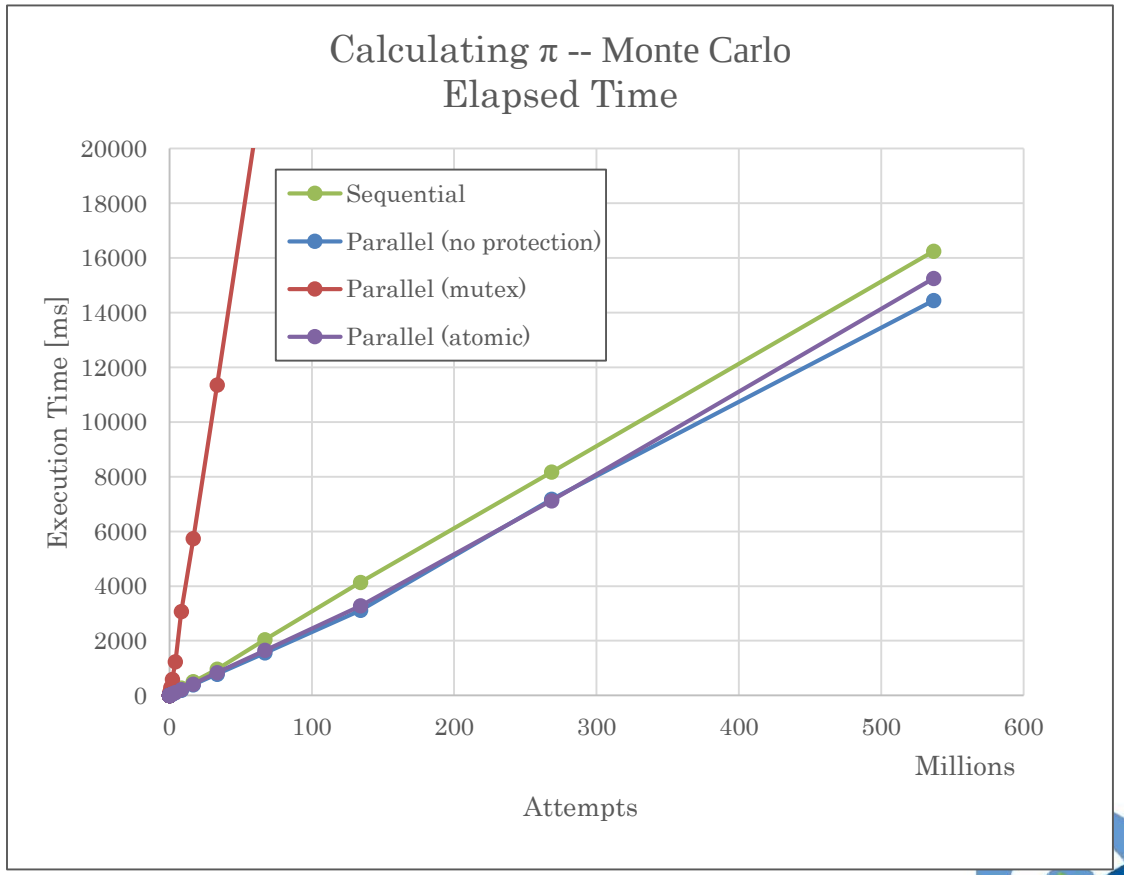


Calculate π in Parallel (atomic)

```
int main() {
    int const n_total = 10000;
    std::atomic<int> nc = 0;

    hpx::experimental::for_loop(hpx::execution::par, 0, n_total,
        [&](int n) {
            double x = get_random_number();
            double y = get_random_number();
            if (sqr(x) + sqr(y) <= 1.0)
                ++nc;
        });

    std::println("Approximated pi for {} attempts: {}",
        n_total, (4.0 * nc) / n_total);
}
```

Calculate π in Parallel (atomic)

Attempts	pi	Error	Elapsed [ms]
2	2	0.363380228	0
4	4	0.273239545	0
8	2.5	0.204225285	0
16	3.25	0.03450713	0
32	3.25	0.03450713	0
64	2.9375	0.064964709	0
128	3.21875	0.024559946	0
256	3.09375	0.01522879	0
512	3.078125	0.020202382	0
1024	3.12890625	0.004038208	0
2048	3.1640625	0.007152374	0
4096	3.1796875	0.012125966	0
8192	3.188476563	0.014923612	0
16384	3.147949219	0.002023358	0
32768	3.152709961	0.003538749	0
65536	3.143981934	0.000760531	1
131072	3.137390137	0.001337703	3
262144	3.144088745	0.000794531	7
524288	3.145782471	0.00133366	13
1048576	3.141014099	0.00018416	30
2097152	3.141271591	0.000102197	57
4194304	3.141586304	2.02E-06	108
8388608	3.142357349	0.00024341	209
16777216	3.141206503	0.000122916	414
33554432	3.141438246	4.91E-05	835
67108864	3.141272724	0.000101837	1650
134217728	3.14153868	1.72E-05	3278
268435456	3.141487181	3.36E-05	7120
536870912	3.141568877	7.57E-06	15258

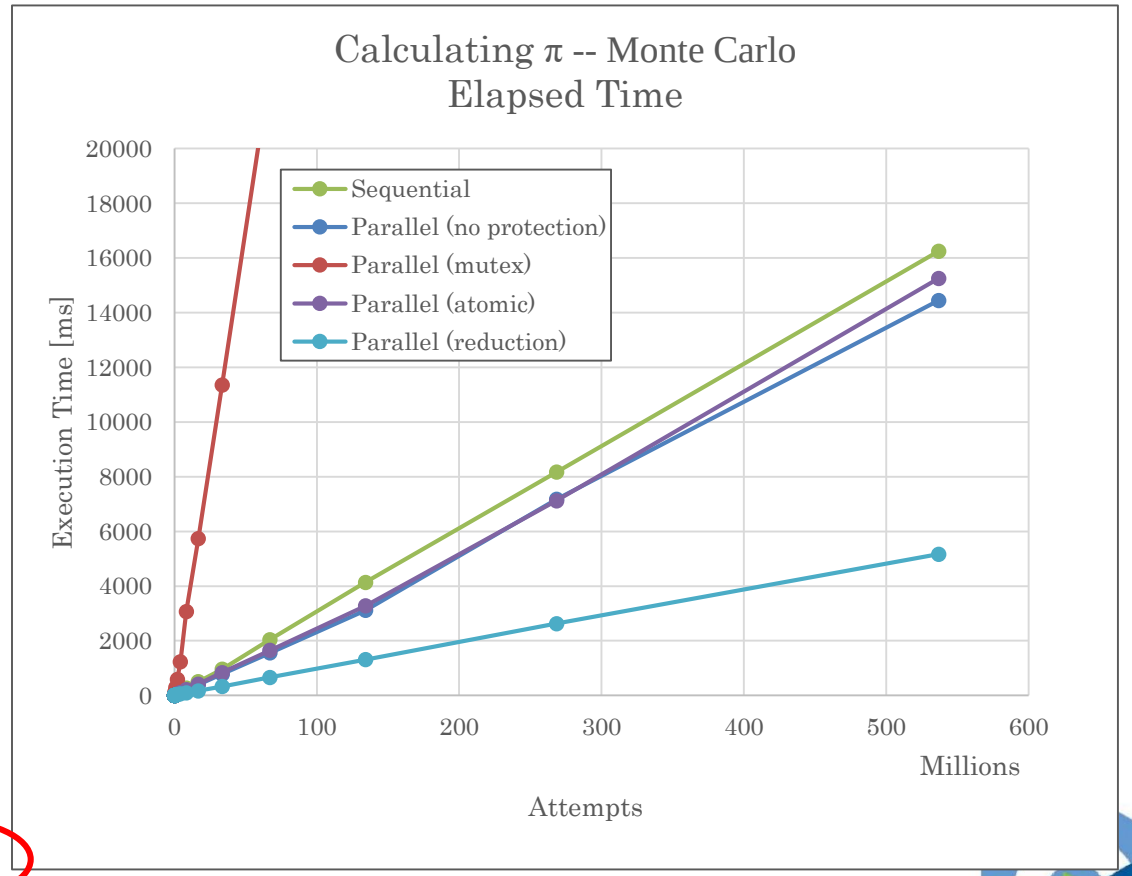


Calculate π in Parallel (reduction)

```
int main() {
    int const n_total = 10000;
    int nc = 0;

    hpx::experimental::for_loop(hpx::execution::par, 0, n_total,
        hpx::experimental::reduction_plus(nc),
        [&](int n, int& nc_local) {
            double x = get_random_number();
            double y = get_random_number();
            if (sqr(x) + sqr(y) <= 1.0)
                ++nc_local;
        });

    std::println("Approximated pi for {} attempts: {}",
        n_total, (4.0 * nc) / n_total);
}
```

Calculate π in Parallel (reduction)

Attempts	pi	Error	Elapsed [ms]
2	2	0.363380228	0
4	3	0.045070341	0
8	3.5	0.114084602	0
16	3.5	0.114084602	0
32	3.25	0.03450713	0
64	3.5625	0.13397897	0
128	2.78125	0.114700629	0
256	3.015625	0.040096749	0
512	3.1171875	0.007768402	0
1024	3.16015625	0.005908976	0
2048	3.126953125	0.004659907	0
4096	3.125	0.005281606	0
8192	3.116699219	0.007923826	0
16384	3.156494141	0.004743291	0
32768	3.157592773	0.005092996	1
65536	3.146850586	0.001673652	1
131072	3.140258789	0.000424582	1
262144	3.143157959	0.000498252	3
524288	3.144371033	0.000884386	7
1048576	3.142715454	0.000357399	30
2097152	3.139610291	0.000631006	34
4194304	3.140385628	0.000384208	57
8388608	3.141095161	0.000158357	93
16777216	3.141840696	7.90E-05	171
33554432	3.141717076	3.96E-05	330
67108864	3.141583681	2.86E-06	662
134217728	3.141607851	4.84E-06	1313
268435456	3.141496509	3.06E-05	2629
536870912	3.141562775	9.51E-06	5170



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP