Fractals

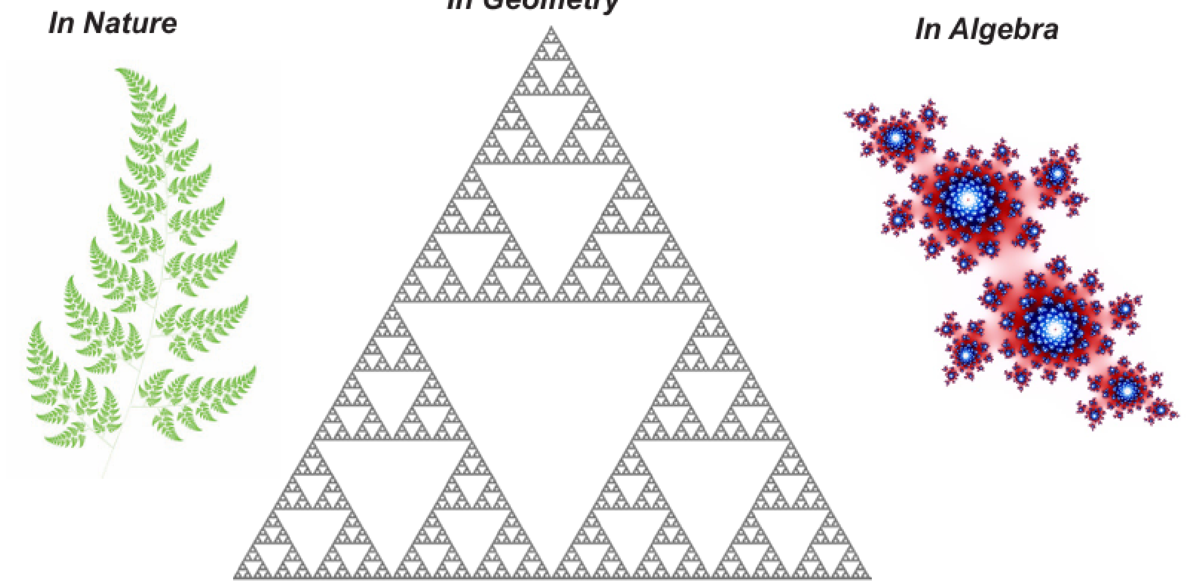
Lecture 7

Hartmut Kaiser

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

What are Fractals?

- A fractal is a never-ending pattern
  - Fractals are infinitely complex patterns that are self-similar across different scales
  - They are created by repeating a simple process over and over in an ongoing feedback loop
- Found everywhere



Fractals in Nature

- Mostly spirals in various dimensions and scales

Scale: ~1m

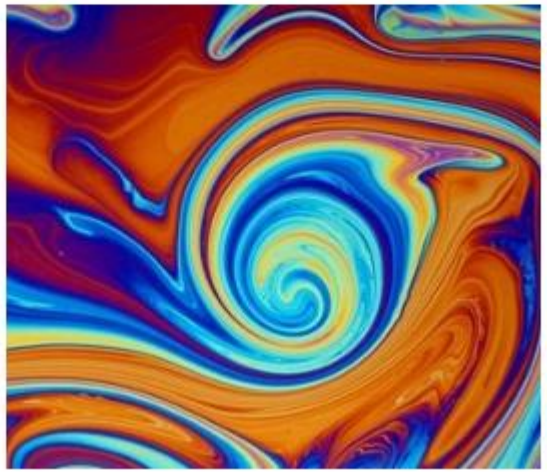
Scale: ~100km

Scale: ~100.000 ly

Fractals in Nature

- Various phenomena

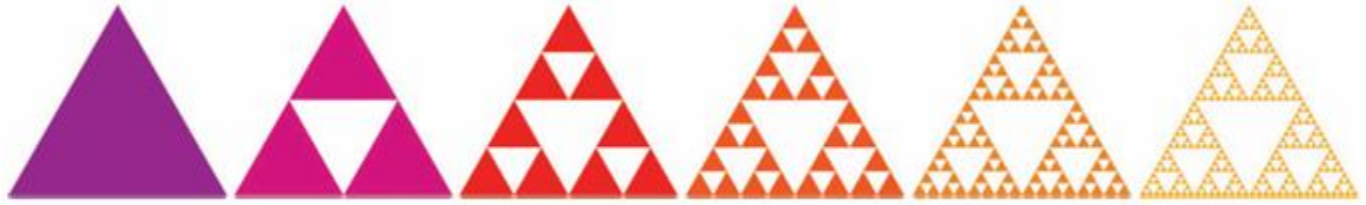
Scale: ~50cm


Scale: ~5mm

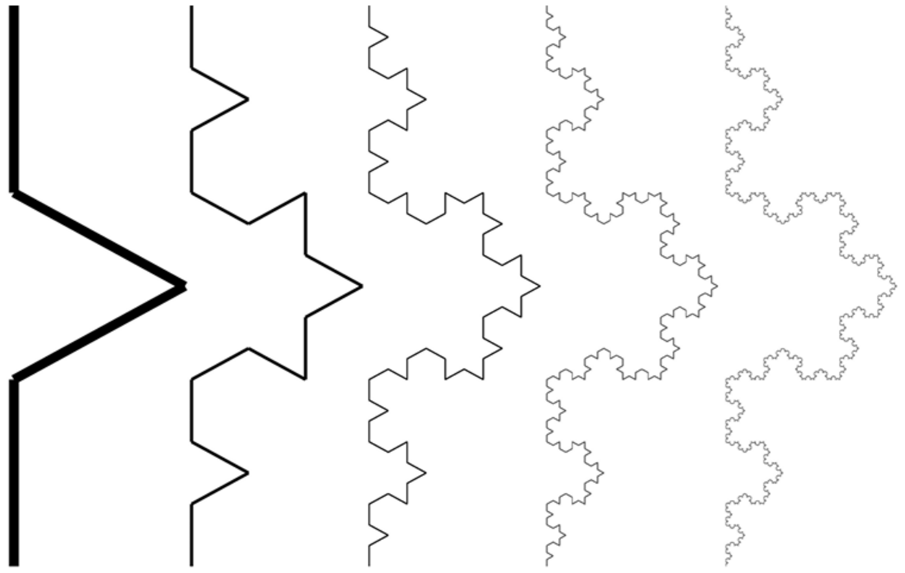
Scale: ~5cm

Geometric Fractals

- Sierpinski Triangle

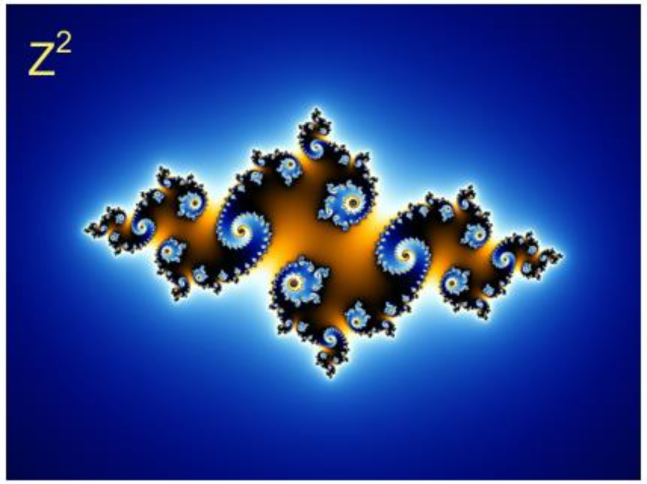


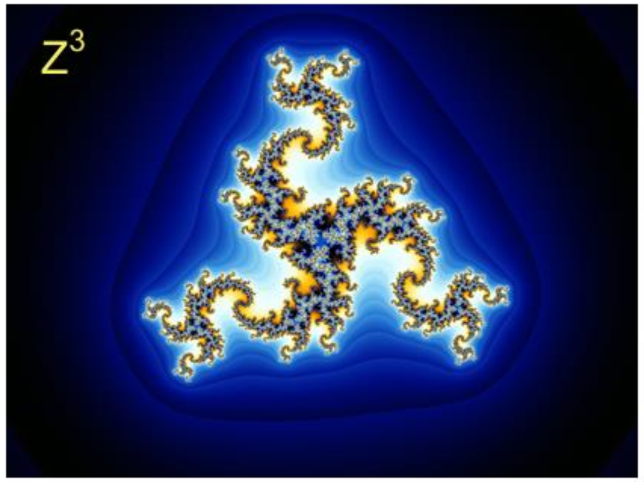
- The Koch Curve

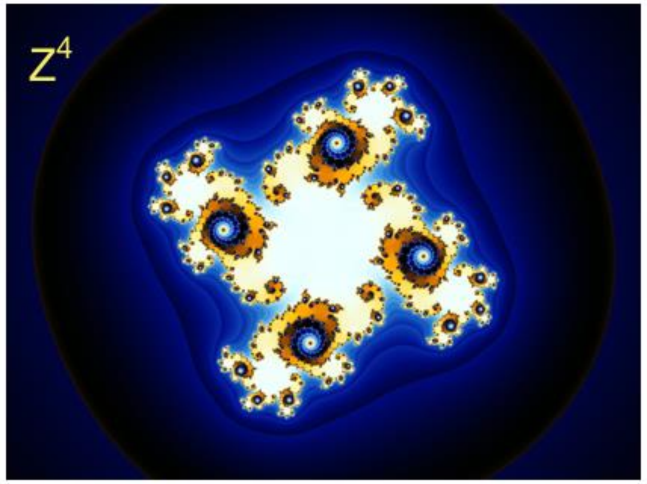


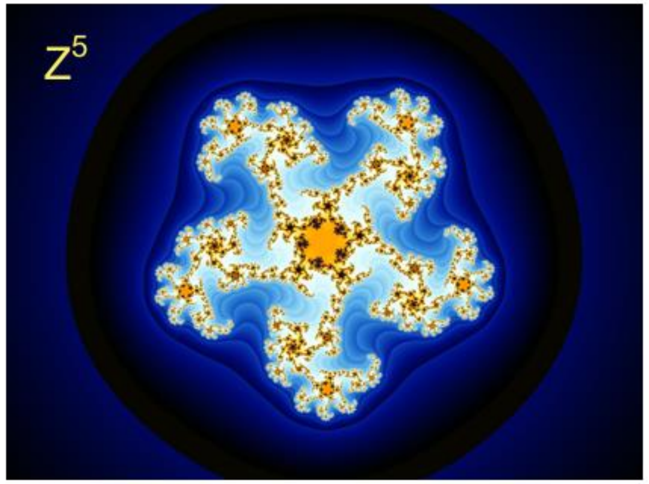
Algebraic Fractals

- The Mandelbrot Set









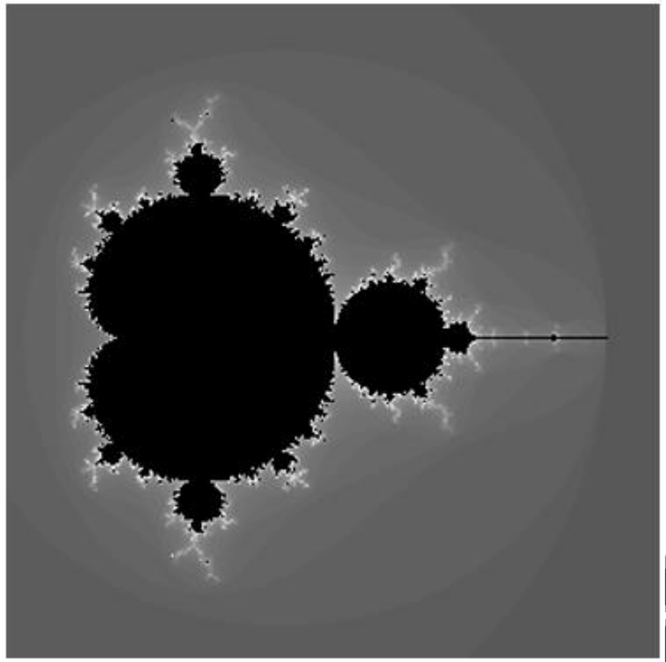
The Mandelbrot Set

The Mandelbrot Set

- Discovered by Benoit Mandelbrot in 1980
- Divides the plane into two regions:
  - An ‘inner region’, the black region in the figure, and an ‘outer region’
  - The boundary between the two regions is a fractal.
- It is defined as the set of complex numbers c for which the function

$$f_c(z) = z^2 + c$$

  does not diverge to infinity when iterated starting at $z = 0$
- IOW, all complex numbers inside the black region



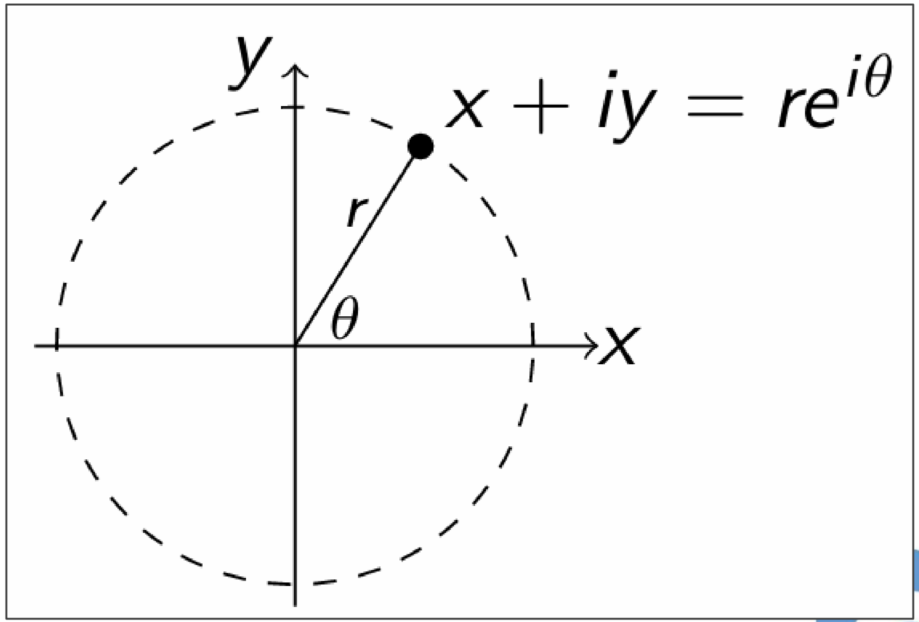
The Complex Plane

$$\mathbb{C} = \{x + iy : x, y \in \mathbb{R}\} = \{re^{i\theta} : r, \theta \in \mathbb{R}\}$$

- Where:

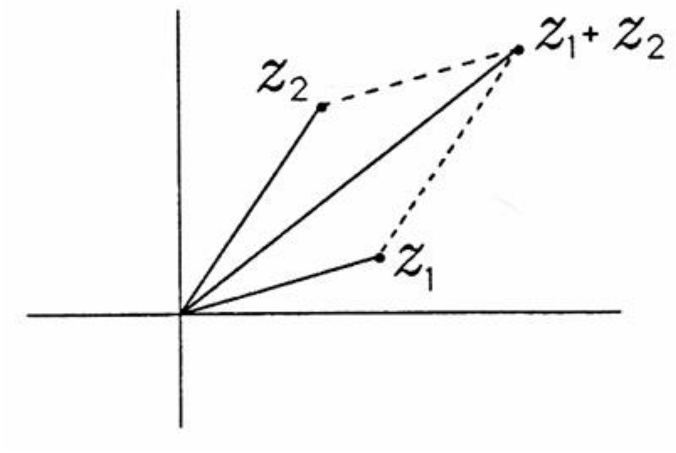
$$re^{i\theta} = (\mathrm{r}\cos\theta) + \mathrm{i}\,(\mathrm{r}\sin\theta)$$

- If $z = x + iy = re^{i\theta} \in \mathbb{C}$, we say
  - x is the **real part** of z
  - y is the **imaginary part** of z
  - $r = |z| = \sqrt{x^2 + y^2}$ is the **modulus** of z
  - θ is the **argument** of z
  - $i = \sqrt{-1}$

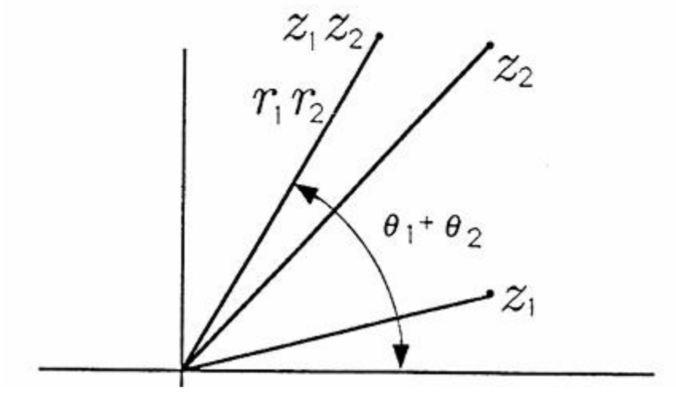


Arithmetic in $\mathbb{C}$

- Complex number addition:
  - Vector style: $z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$



- Complex number multiplication:
  - Multiply moduli; add arguments:
  - $z_1 z_2 = (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + x_2 y_1) = r_1 r_2 e^{i(\theta_1 + \theta_2)}$



Aside: Complex Numbers in C++

- C++ has special type: `std::complex<double>`
  - The type has operators overloaded: `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `>`, …
  - Additional functions: `real`, `imag`, `arg`, `norm`, `conj`, …
  - Has mathematical functions overloaded: `abs`, `sqrt`, `pow`, `exp`, `log`, trigonometric functions, etc.
- To use those, you must add the header `#include <complex>`

Aside: Complex Numbers in C++

```
std::complex<double> num1(3.0, 4.0);               // 3 + 4i
std::complex<double> num2(1.0, 2.0);               // 1 + 2i

std::complex<double> sum = num1 + num2;            // Adding complex numbers
std::complex<double> difference = num1 - num2; // Subtract them
std::complex<double> product = num1 * num2;        // Multiply them
std::complex<double> quotient = num1 / num2;       // Divide them


std::cout << "Sum: " << sum << std::endl;                   // Sum: (4,6)
std::cout << "Difference: " << difference << std::endl; // Difference: (2,2)
std::cout << "Product: " << product << std::endl;           // Product: (-5,10)
std::cout << "Quotient: " << quotient << std::endl;         // Quotient: (2.2,-0.4)
```

Aside: Complex Numbers in C++

```
// defines the complex number: (10 + 2i)
std::complex<double> c(10.0, 2.0);

// prints the real part using the real function
std::cout << "Real part: " << std::real(c) << std::endl;
std::cout << "Imaginary part: " << std::imag(c) << std::endl;

// prints the absolute value of the complex number
std::cout << "The absolute value of " << c << " is: ";
std::cout << std::abs(c) << std::endl;

// use of norm()
std::cout << "The norm of " << c << " is " << std::norm(c) << std::endl;

// prints the argument of the complex number
std::cout << "The argument of " << c << " is: ";
std::cout << std::arg(c) << std::endl;

// use of polar()
std::cout << "The complex whose magnitude is " << 2.0;
std::cout << " and phase angle is " << 0.5;
std::cout << " is " << std::polar(2.0, 0.5) << std::endl;
```

Aside: Complex Numbers in C++

```
// initializing the complex: (-1.0 + 0.0i)
std::complex<double> c(-1.0, 0.0);

// use of cos(): (1.54308, 0)
std::cout << "The cos of " << c << " is " << std::cos(c) << std::endl;

// use of sin(): (0, 1.1752)
std::cout << "The sin of " << c << " is " << std::sin(c) << std::endl;

// use of tan(): (0, 0.761594)
std::cout << "The tan of " << c << " is " << std::tan(c) << std::endl;
```

Aside: Complex Numbers in C++

- Main takeaway:
  - `std::complex<double>` is a `Regular` type (even `TotallyOrdered`)
  - Use `std::complex<double>` as if it was a `double`
- Demonstrates one of the (many) powers of C++
  - One can customize the operations for any type to operate as needed
- Operator overloading in C++:

```
struct my_complex {
    double real, imag;
    my_complex(double r, double i = 0.0) : real(r), imag(i) {}

    friend my_complex operator+(my_complex lhs, my_complex rhs) {
        return {lhs.real + rhs.real, rhs.imag + rhs.imag};
    }
};
```

The Mandelbrot Set

The Mandelbrot Set

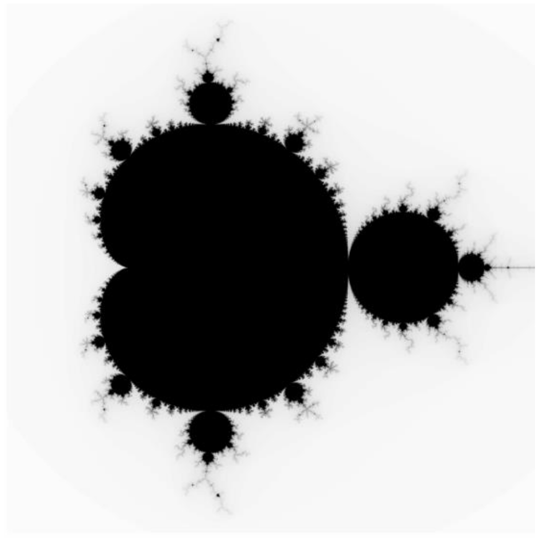
- Given:

$$f_c(z) = z^2 + c$$

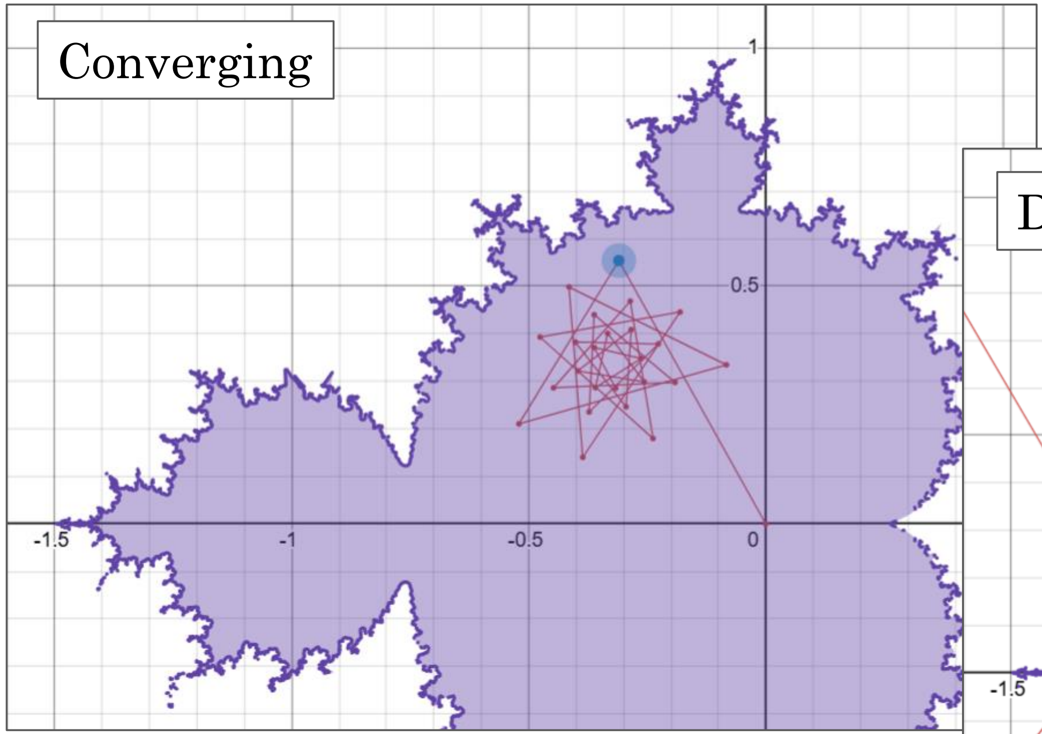
- Then the **Mandelbrot Set** is

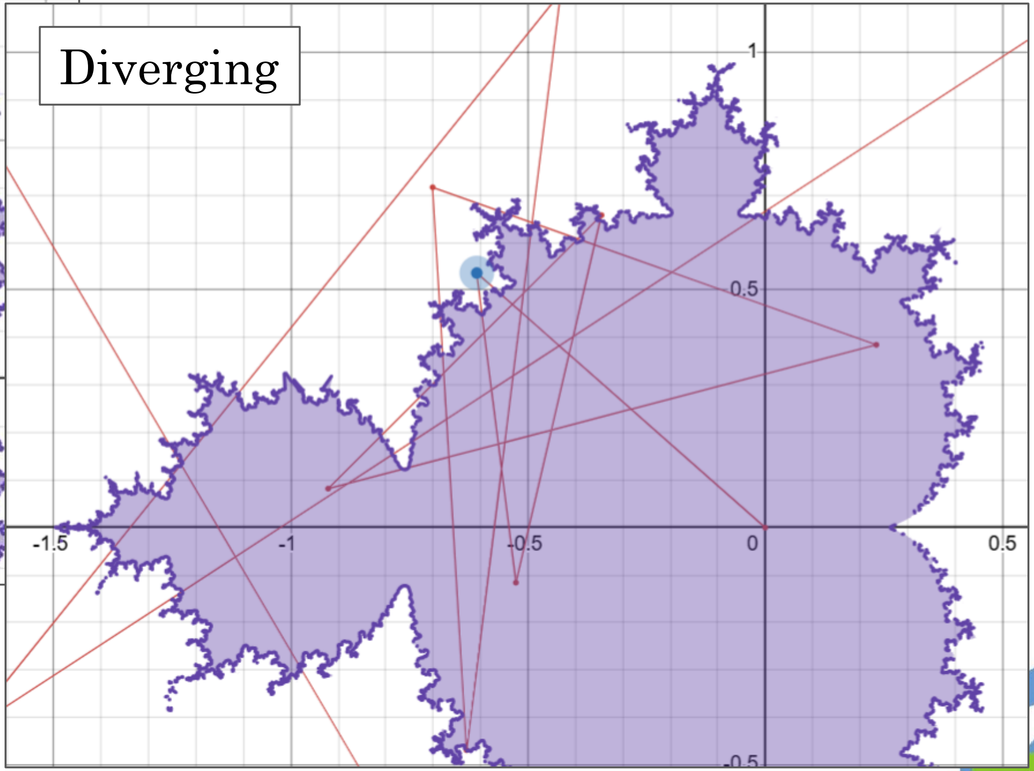
$$\mathbb{M} = \{c \in \mathbb{C}: \{f_c^n(0): n \geq 1\}\ is\ bounded\}$$

- (Benoit Mandelbrot, 1980)
- IOW, if the orbit created by $f_c^n(0)$ is **not** diverging for $c \in \mathbb{C}$ then c is part of the Mandelbrot Set

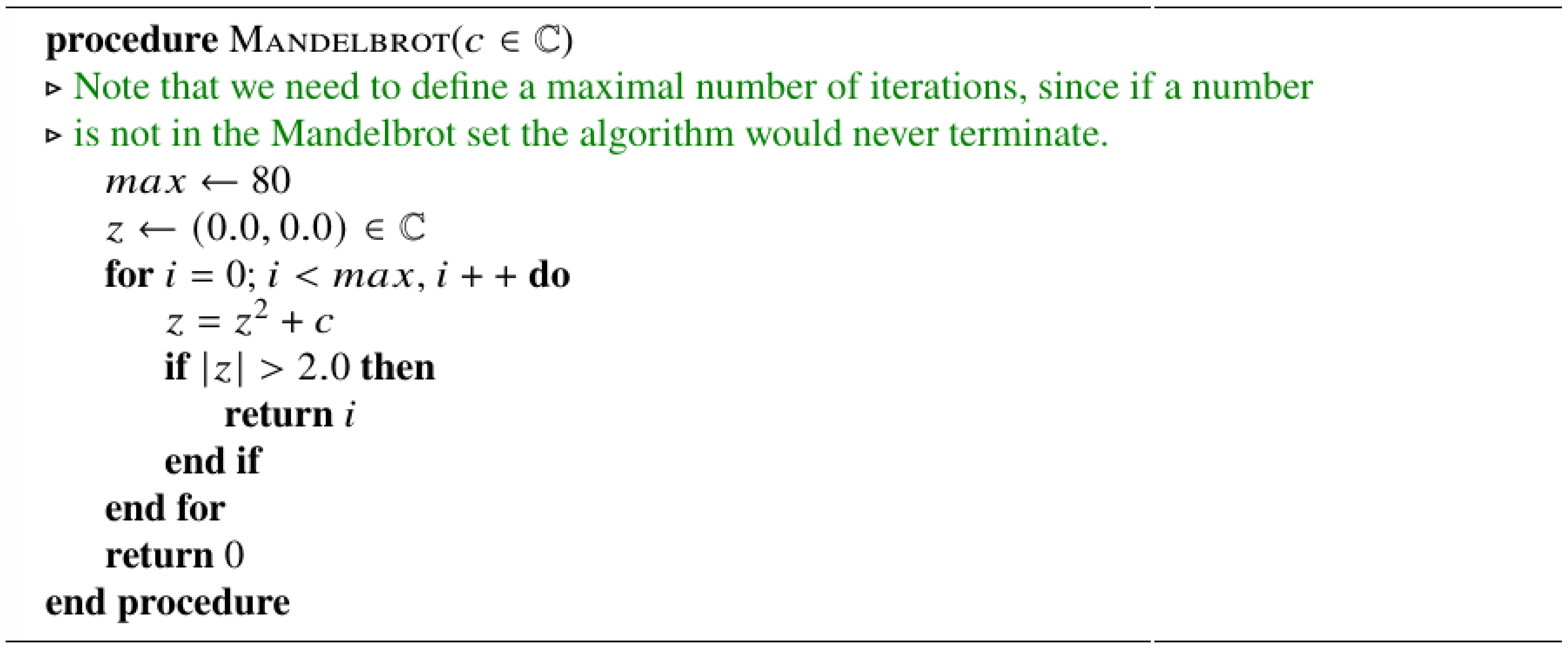


The Mandelbrot Set: Trajectories





The Mandelbrot Set: The Algorithm



```
procedure MANDELBROT($c \in \mathbb{C}$)
▷ Note that we need to define a maximal number of iterations, since if a number
▷ is not in the Mandelbrot set the algorithm would never terminate.
    $max \leftarrow 80$
    $z \leftarrow (0.0, 0.0) \in \mathbb{C}$
    for $i = 0; i < max, i++$ do
        $z = z^2 + c$
        if $|z| > 2.0$ then
            return $i$
        end if
    end for
    return 0
end procedure
```

The Mandelbrot Set: The Code

```
size_t mandelbrot(std::complex<double> c)
{
    size_t const max_iterations = 100;

    std::complex<double> z(0.0, 0.0);
    for (size_t i = 0; i != max_iterations; ++i)
    {
        z = z * z + c;
        if (std::abs(z) > 2.0)
            return i;   // diverging
    }
    return -1;          // not diverging, part of the Mandelbrot Set
}
```

The Mandelbrot Set: The Code

```cpp
template <typename Data>
std::pair<Data, int> fixed_point(
    Data f(Data), bool cond(Data, Data), int n, Data init)
{
    Data x = init;
    for (int i = 0; i != n; ++i) {
        Data x1 = f(x);      // get next iteration result
        if (cond(x, x1))     // compare with previous iteration result
            return {x1, i};
        x = x1;              // update for next iteration
    }
    return {x, n};
}
```

The Mandelbrot Set: The Code

```
size_t mandelbrot(std::complex<double> c)
{
    size_t const max_iterations = 100;
    using float_type = std::complex<double>;

    auto [_, iterations] =      // unpack returned std::pair
        fixed_point(
            // function to find fixed point for
            [c](float_type z) { return z * z + c; },
            // termination condition
            [](float_type, float_type z) { return std::abs(z) > 2.0; },
            max_iterations, float_type(0, 0));

    return iterations == max_iterations ? -1 : iterations;
}
```

The Mandelbrot Set: The Code

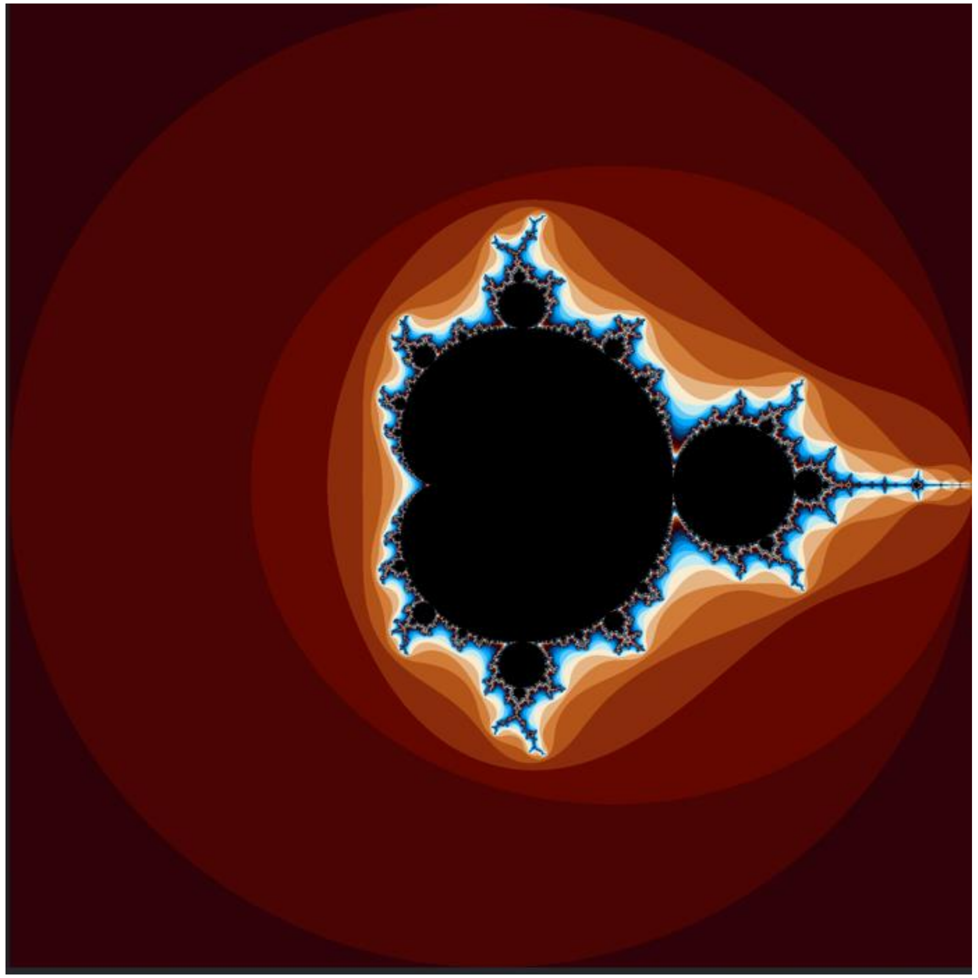
```
std::for_each(counting_iterator(0), counting_iterator(size_y),
     [&](size_t pixel_y) {
        double imag = scale(pixel_y, 0, size_y, -2.0, 2.0);
        for (size_t pixel_x = 0; pixel_x != size_x; ++pixel_x) {
            // Get the number of iterations (-1 means not diverging)
            double real = scale(pixel_x, 0, size_x, -2.0, 2.0);
            int value = mandelbrot(std::complex(real, imag));
            if (value == -1)
                // Not diverging, part of Mandelbrot Set
                mandelbrot_img.SetPixel(pixel_x, pixel_y, RGBApixel(0, 0, 0));
            else
                // Convert the value to RGB color space and set the pixel color
                mandelbrot_img.SetPixel(pixel_x, pixel_y, get_rgb(value));
        }
    });
```

The Mandelbrot Set: The Code

```
template <typename T1, typename T2>
T2 scale(T1 value, T1 min_value, T1 max_value, T2 min_range, T2 max_range)
{
    return (max_range - min_range) / T2(max_value - min_value) *
        T2(value) - min_range;
}
```

- Simple scaling of `value` (that is of range `[min_value, max_value)`) to output range `[min_range, max_range)`

The Mandelbrot Set: The Result



The Mandelbrot Set: Parallelize

- Computation for each pixel is
  - Independent from computation of any other pixel
  - Sequence of computations is irrelevant
  - No inter-dependencies between pixels
- Embarrassingly parallel
  - Decomposition is 'embarrassingly' trivial
  - Relatively easy to parallelize
- Fork-join parallelism as all pixels have to be available before image is done
  - Fork: start multiple tasks (threads)
  - Join: wait for all tasks (threads) to finish

Fork-Join Parallelism

- Used for years: OpenMP, CILK, Java concurrency Framework, Task Parallel Library for .NET

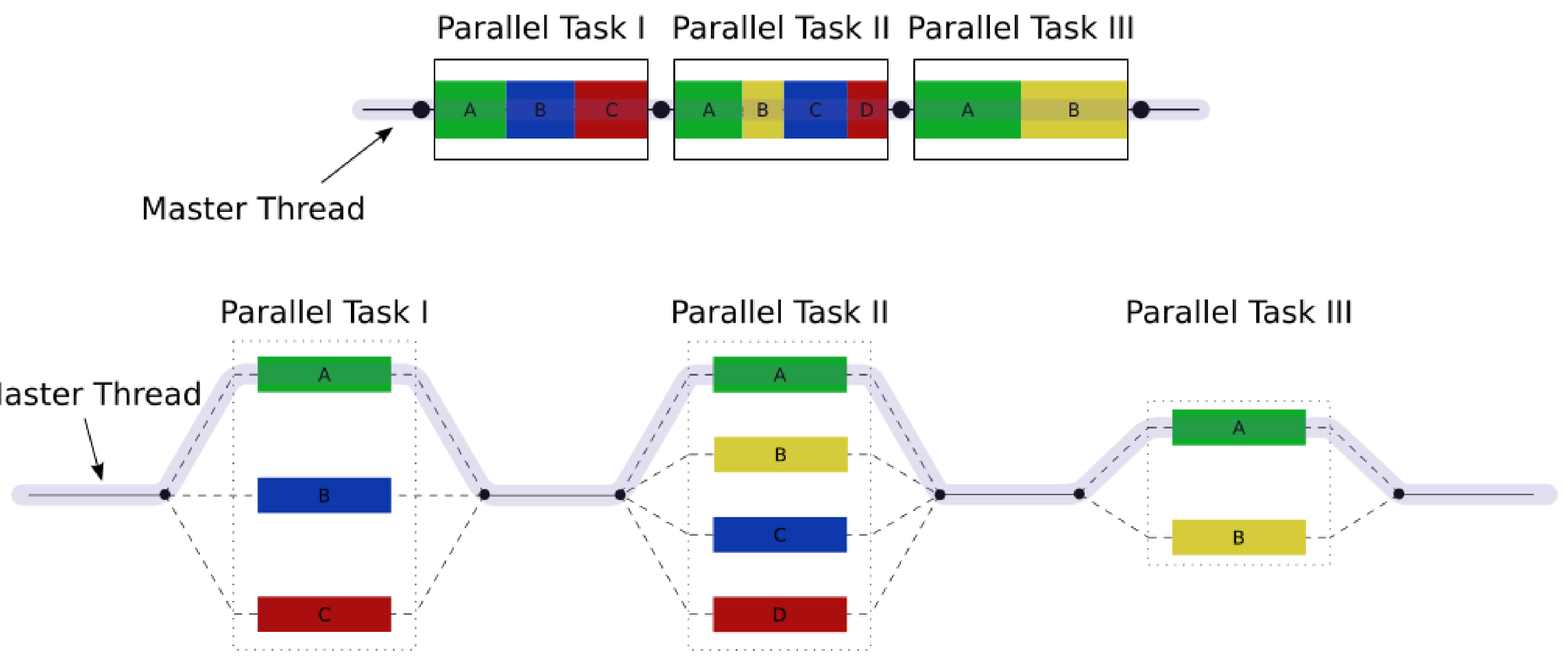
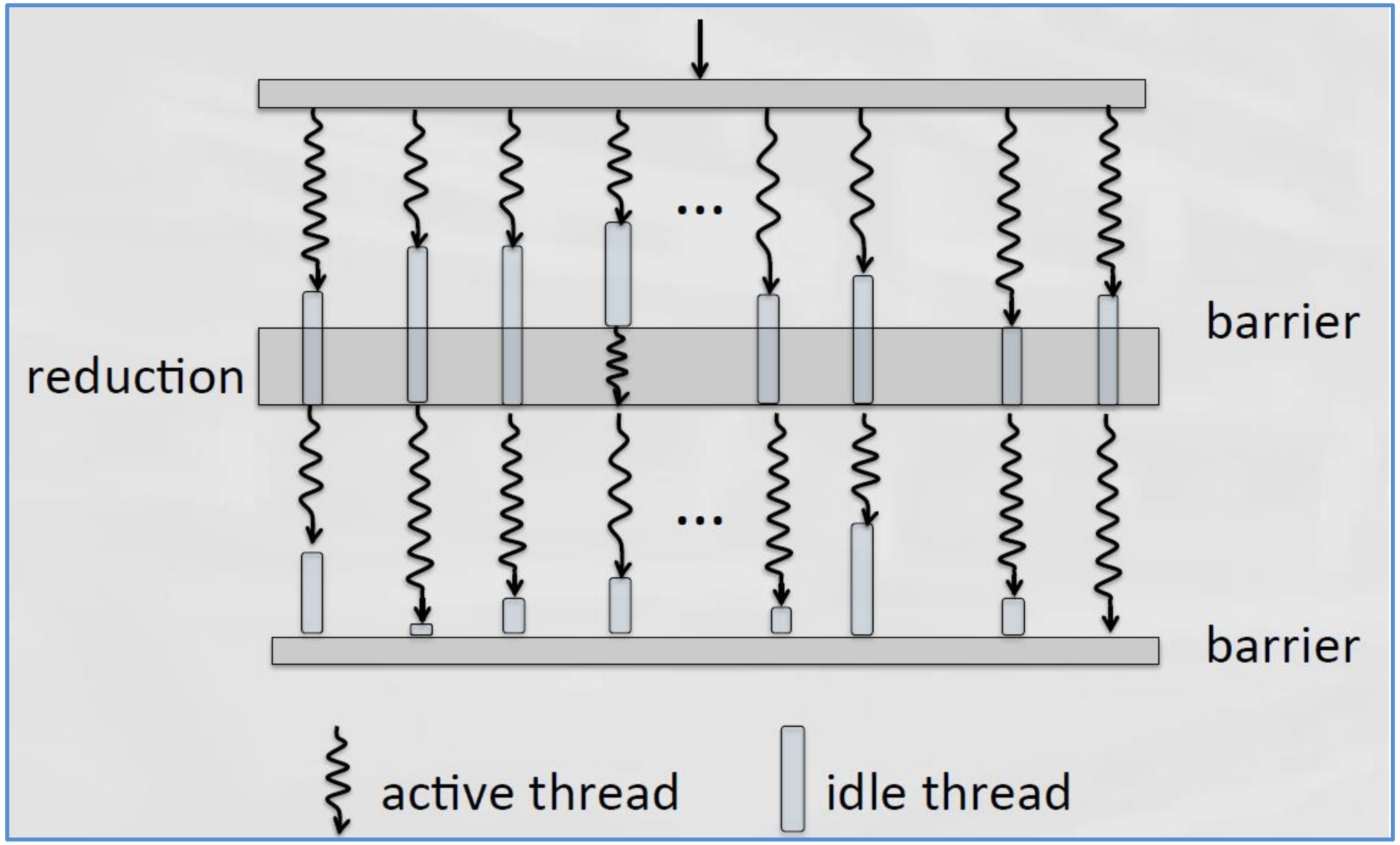


Image courtesy of: Wikipedia: http://en.wikipedia.org/wiki/Fork%E2%80%93join_model

Fork-Join Parallelism



The Mandelbrot Set: Parallelize

- Computation for each pixel is
  - Independent from computation of any other pixel
  - Sequence of computations is irrelevant
  - No inter-dependencies between pixels
- Embarrassingly parallel
  - Relatively easy to parallelize
- Fork-join parallelism as all pixels have to be available before image is done
- C++ has parallel algorithms that are suitable for this kind of problems
  - Loop iterations are independent, no sequencing is required
  - `std::for_each(...)` → `std::for_each(std::execution::par, ...)`

C++ Parallel Algorithms

- Mostly, same semantics as sequential algorithms
  - Additional, first argument: `execution_policy`
  - Defined in namespace `std::execution`

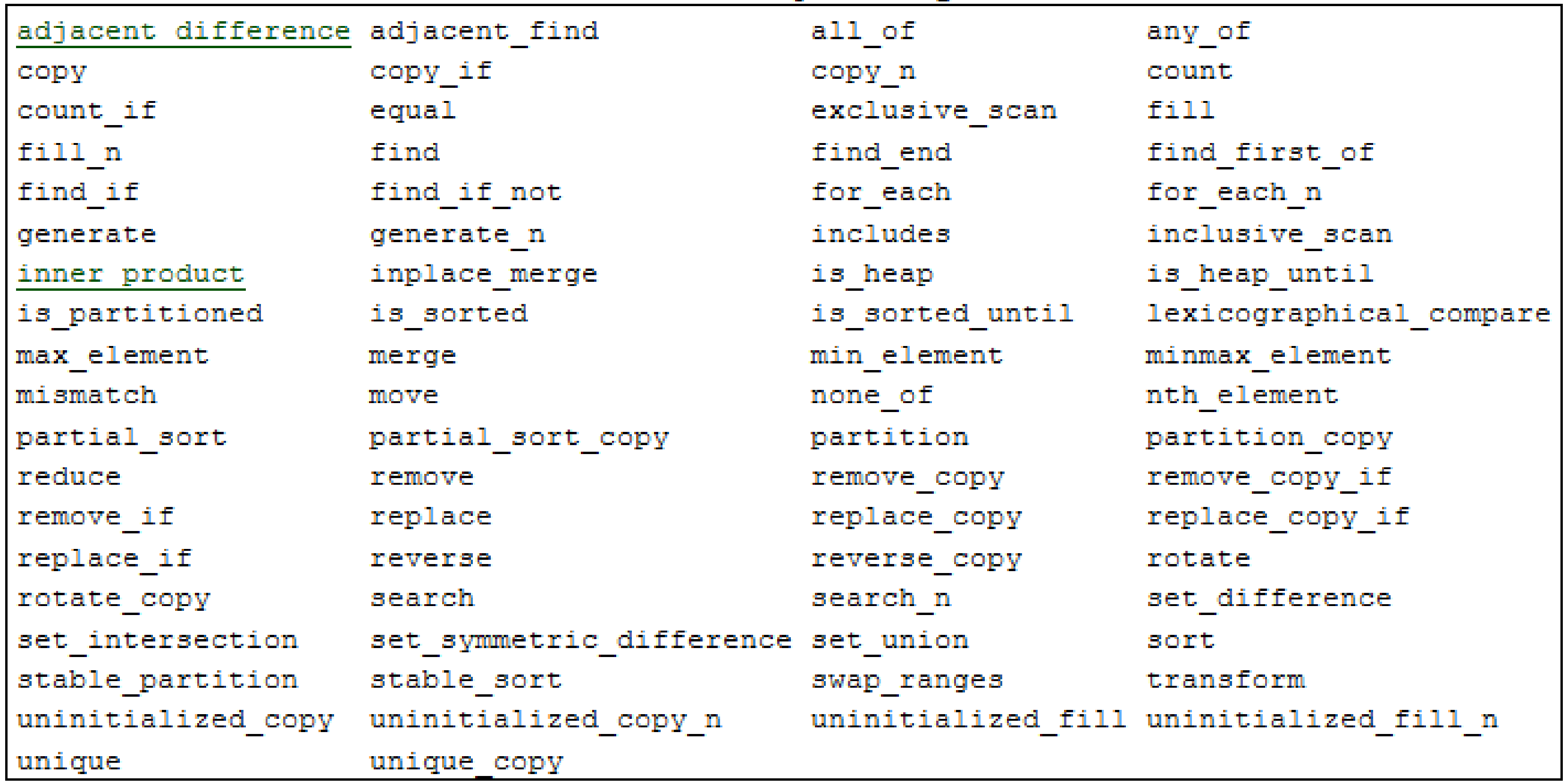
• `sequenced_execution_policy`:	`seq`
• `parallel_execution_policy`:	`par`
• `parallel_unsequenced_execution_policy`:	`par_unseq`
• `unsequenced_execution_policy`:	`unseq`

- Entirely fork-join as algorithms return only after all work has been done
  - Performance of those algorithms depends on high quality schedulers

C++ Parallel Algorithms

- `sequenced_execution_policy` (`seq`)
  - Iterations must be executed in-order on calling thread, no re-ordering allowed
- `parallel_execution_policy (par)`
  - Iterations can be re-ordered and can be run on arbitrary threads
  - (parallelization is allowed)
- `parallel_unsequenced_execution_policy (par_unseq)`
  - Iterations may be parallelized, vectorized, or migrated across threads
  - Special rules related to exception handling
- `unsequenced_execution_policy` (`unseq`)
  - Iterations may be vectorized, e.g., executed on a single thread using instructions that operate on multiple data items.
  - Special rules related to exception handling

C++ Parallel Algorithms



adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
uninitialized_copy	uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n
unique	unique_copy		

The Mandelbrot Set: The Code

```
std::for_each(std::execution::par,
  counting_iterator(0), counting_iterator(size_y),
  [&](size_t pixel_y) {
      double imag = scale(pixel_y, 0, size_y, -2.0, 2.0);
      for (size_t pixel_x = 0; pixel_x != size_x; ++pixel_x) {
          // Get the number of iterations (-1 means not diverging)
          double real = scale(pixel_x, 0, size_x, -2.0, 2.0);
          int value = mandelbrot(std::complex(real, imag));
          if (value == -1)
              // Not diverging, part of Mandelbrot Set
              mandelbrot_img.SetPixel(pixel_x, pixel_y, RGBApixel(0, 0, 0));
          else
              // Convert the value to RGB color space and set the pixel color
              mandelbrot_img.SetPixel(pixel_x, pixel_y, get_rgb(value));
      }
 });
```

The Mandelbrot Set: The Code

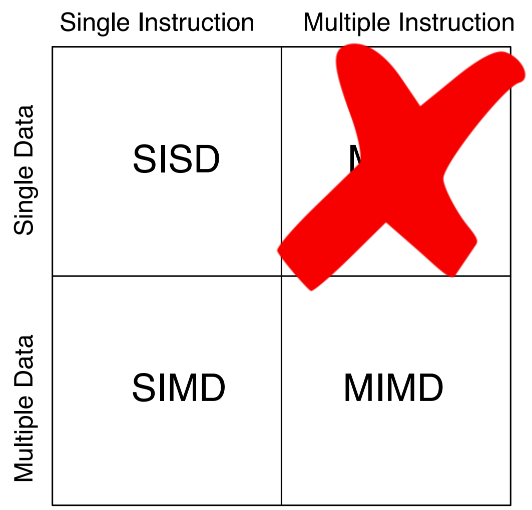
```
hpx::experimental::for_loop(hpx::execution::par,
    0, size_y,
    [&](size_t pixel_y) {
       double imag = scale(pixel_y, 0, size_y, -2.0, 2.0);
       for (size_t pixel_x = 0; pixel_x != size_x; ++pixel_x) {
           // Get the number of iterations (-1 means not diverging)
           double real = scale(pixel_x, 0, size_x, -2.0, 2.0);
           int value = mandelbrot(std::complex(real, imag));
           if (value == -1)
               // Not diverging, part of Mandelbrot Set
               mandelbrot_img.SetPixel(pixel_x, pixel_y, RGBApixel(0, 0, 0));
           else
               // Convert the value to RGB color space and set the pixel color
               mandelbrot_img.SetPixel(pixel_x, pixel_y, get_rgb(value));
       }
   });
```

Exercise

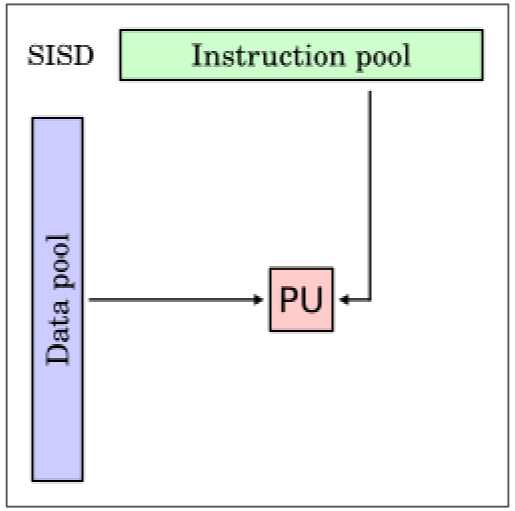
- Implement generating **Julia Set** of your choosing for different values of c
  - Similar to the Mandelbrot Set (which fixes $z_0 = 0$ and plots the number of iterations for all complex values c)
  - Julia set instead fixes c to some number and plots all complex values z_0
- Example values for c that generate beautiful images:
  - $c = -0.8 + 0.156i$
  - $c = -0.7269 + 0.1889i$
  - $c = 0.285 + 0.01i$
- This produces a very similar workload to the Mandelbrot set, but produces a completely different image

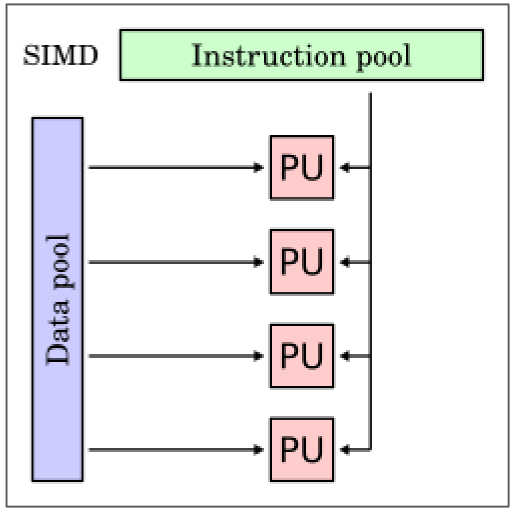
Flynn's Taxonomy (Aside)

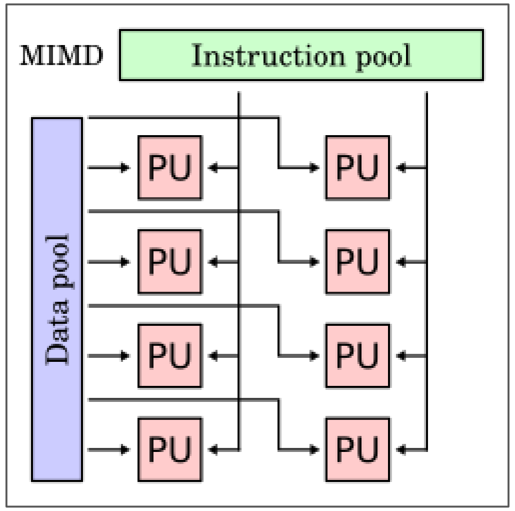
- Classic classification of parallel architectures (Michael Flynn, 1966)
- Based on multiplicity of instruction streams, data storage
  - SISD: plain old sequential
  - SIMD: vectorization
  - MIMD: conventional multi-threading



Flynn's Taxonomy (Aside)

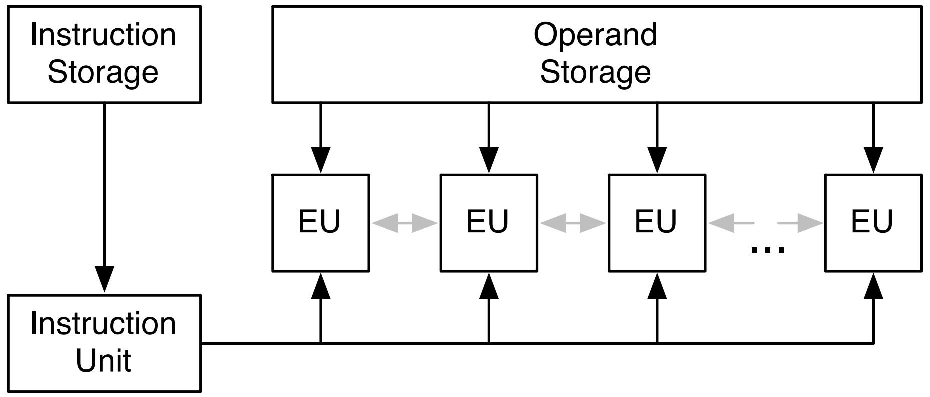






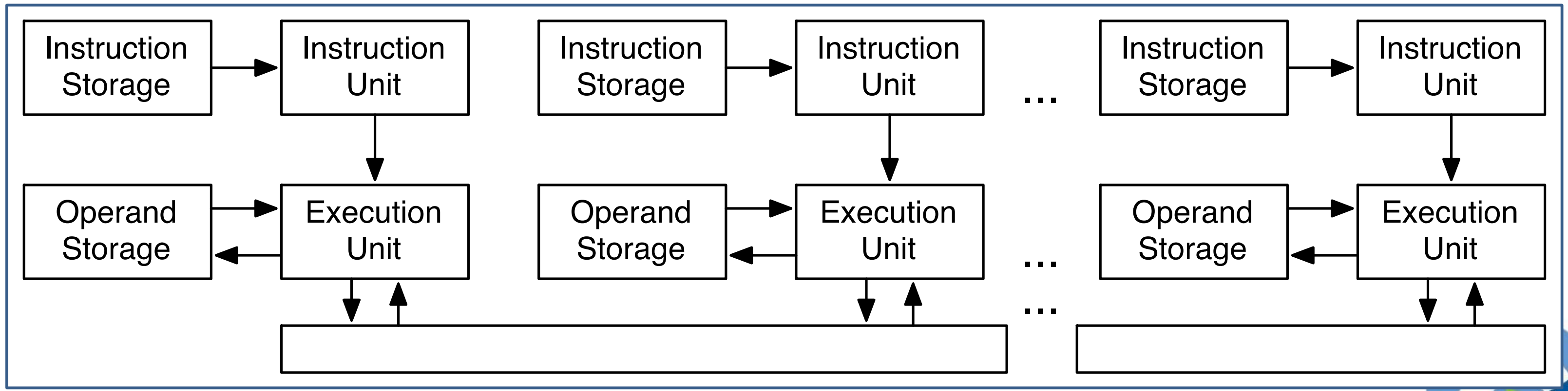
SIMD and MIMD

- Two principal parallel computing paradigms (multiple operands)
- SIMD:
  - Multiple instructions at a time
  - All execution units execute in (c)lock step
  - But each have their own data



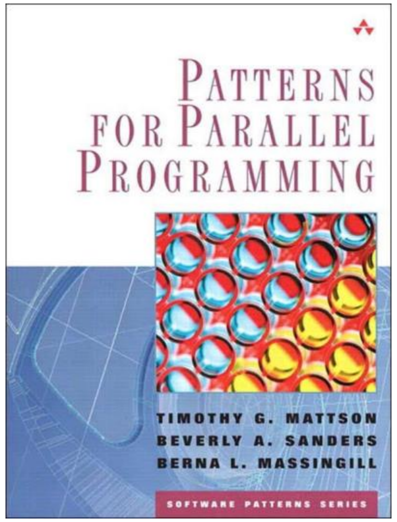
SIMD and MIMD

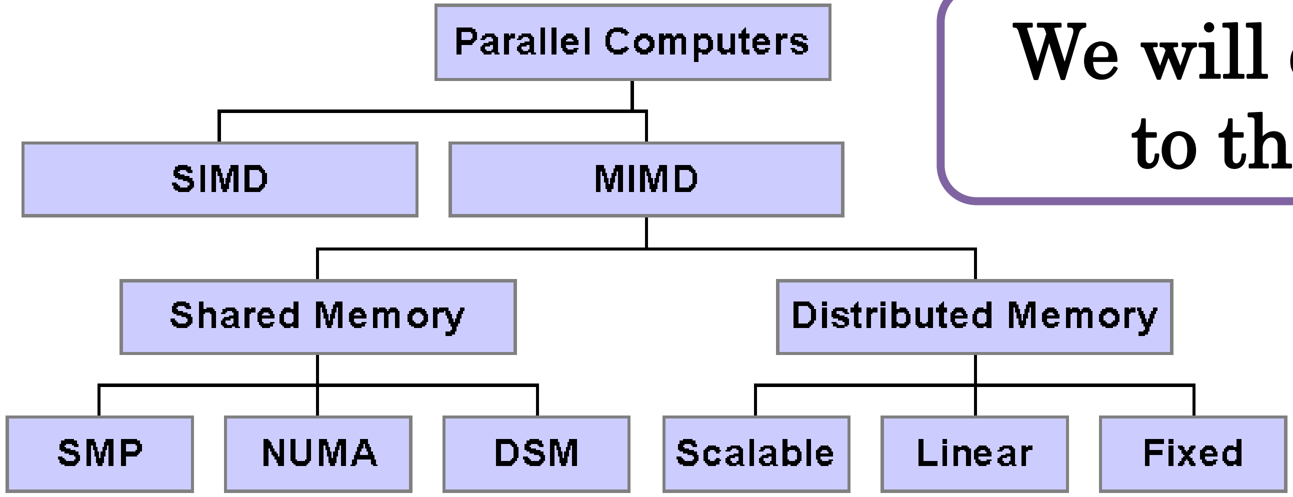
- Two principal parallel computing paradigms (multiple operands)
- MIMD:
  - Single instruction at a time
  - All execution units run independently (with own instructions)
  - Shared memory (same computational node)
  - Memory not shared (distributed computing, coming soon)



A More Refined (Programmer-Oriented) Taxonomy

- Three major modes: SIMD, Shared Memory, Distributed Memory
- Different programming approaches are generally associated with different modes of parallelism
  - HPX allows to unify all of those
- A modern supercomputer will have all three major modes present





We will come back to this soon

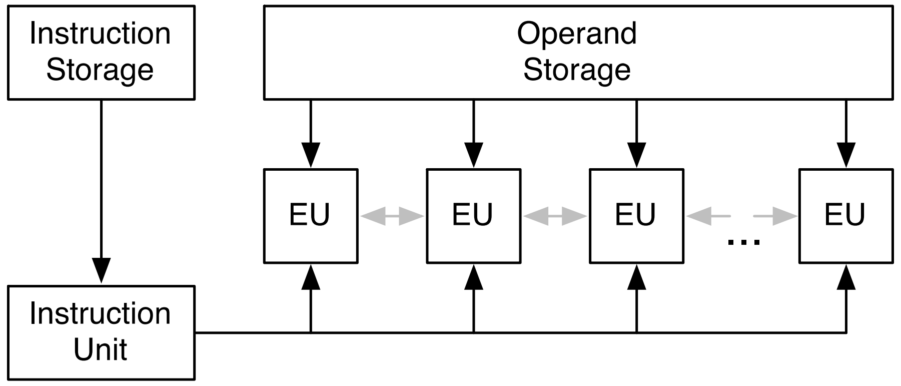
http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/ParallelHardware.htm

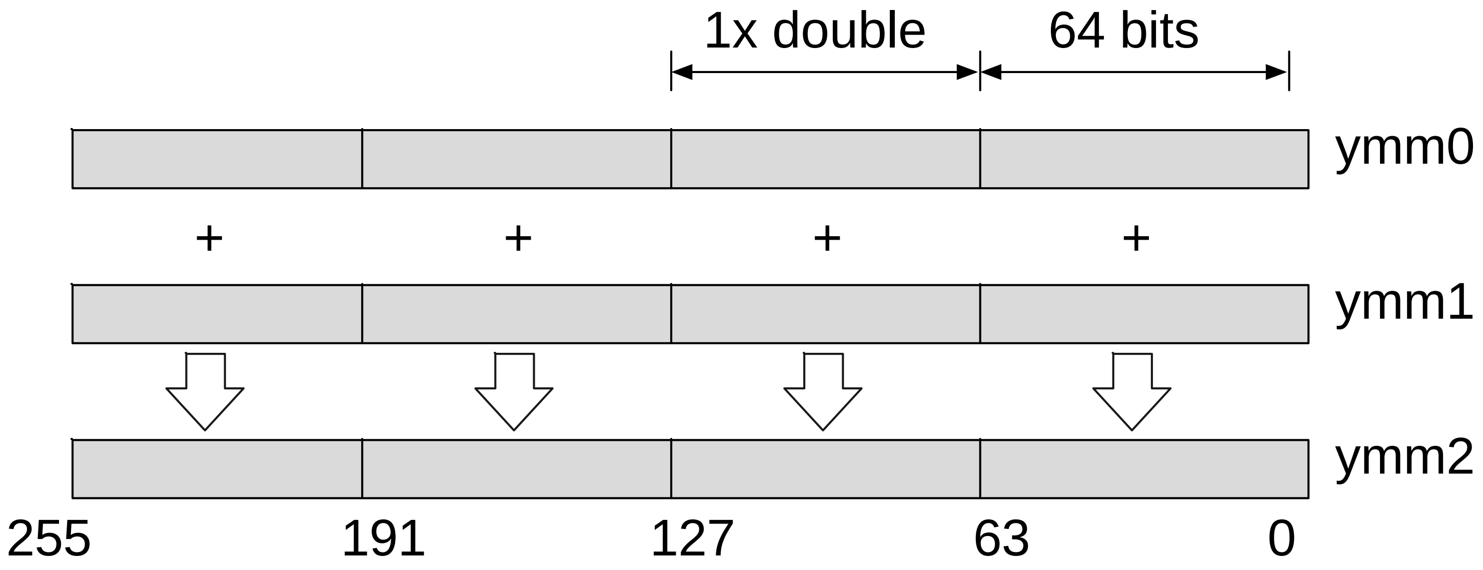
SIMD in SSE/AVX

- One machine instruction

```
vfadd231pd  %ymm0, %ymm1, %ymm2
```

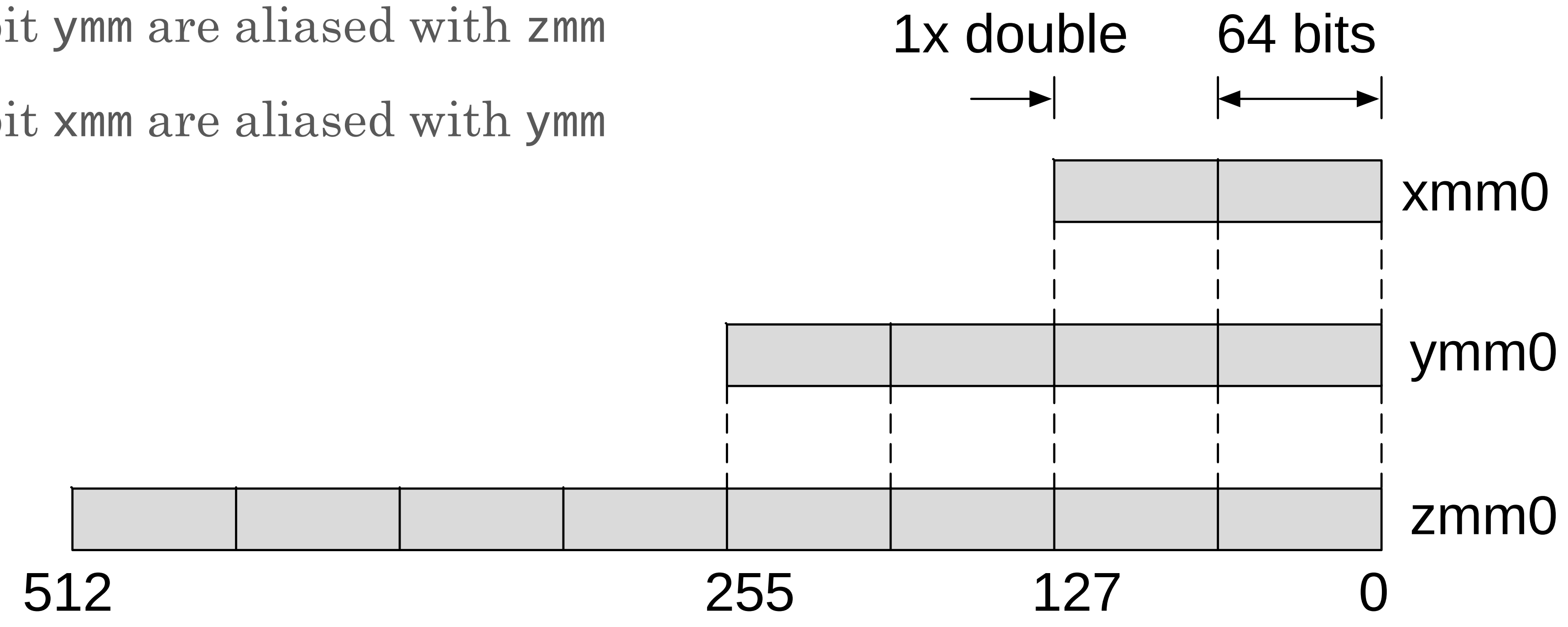
- Adds four doubles simultaneously
- ymm are 256 bit registers





Vector Registers (x86)

- `zmm` are 512 bit registers
- 256 bit `ymm` are aliased with `zmm`
- 128 bit `xmm` are aliased with `ymm`



Recommendations

- Avoid programming in assembler, or intrinsics
  - Non-portable, hardware-specific
- It is most important to have a mental model for the vector registers and to be aware of what is possible and how to write code to be optimizable
- Check your performance against performance models
- Monitor what your compiler is doing
  - Optimization report
  - Full set of flags
  - Last resort – read the assembler

Vectorization in C++

- C++26 introduces `std::simd<T>`, a data type that represents a vector register
  - Here `T` is the type to vectorize over, e.g., `std::simd<double>`
  - For us (C++23), this type exists as `std::experimental::simd<T>`
  - `T` can be any integral or floating point type (depends on architecture)
- Provides portable types
  - For explicitly stating data-parallelism and
  - Structuring data for more efficient SIMD access
- An object of type `simd<T>` behaves analogous to objects of type `T`
  - `T` stores and manipulates one value
  - `simd<T>` stores and manipulates multiple values
- All operators and operations on `simd<T>` act element-wise
  - Well, except for horizontal operations, like `reduce`

Vectorization in C++

```
namespace stdx = std::experimental;

int main() {
    stdx::simd<int> a = 1;                              // uniform initialization
    print_simd("a: ", a);                               // prints: a: 1 1 1 1

    stdx::simd<int> b([](int i) { return i - 2; });
    print_simd("b: ", b);                               // prints: b: -2 -1 0 1

    print_simd("c: {}", a + b);                         // arithmetic operations
    print_simd("d: ", my_abs(c));                       // prints: d: 1 0 1 2

    auto inner_product = stdx::reduce(c);               // horizontal reduction
    print_simd("inner product: ", inner_product); // prints: inner product: 2

    stdx::simd<long double> x([](int i) { return i; });
    print_simd("cos²(x) + sin²(x): ", pow(cos(x), 2) + pow(sin(x), 2));
}
```

Vectorization in C++

- For completeness sake:

```
template <typename T>
void print_simd(std::string const& name, stdx::simd<T> const& a)
{
    std::print(name);
    for (std::size_t i = 0; i != std::size(a); ++i)
        std::print("{} ", a[i]);
    std::println();
}
```

SIMD Conditionals

- This doesn't work (why?):

```
template <typename T>
stdx::simd<T> my_abs(stdx::simd<T> x)
{
    if (x < 0) x = -x;
    return x;
}
```

- Separate facility:

```
template <typename T>
stdx::simd<T> my_abs(stdx::simd<T> x)
{
    stdx::simd_select(x < 0, x) = -x;
    return x;
}
```

SIMD Masks

- The class template `stdx::simd_mask` is a data-parallel type with the element type `bool`

- Relational operators for `stdx::simd` return a `stdx::simd_mask`
  - Each of the mask's elements represent the result of the relational operator for the corresponding element
  - E.g.,

    ```
    // every boolean in m represents result for one element of x
    stdx::simd_mask m = x < 0;
    ```

  - `stdx::simd_select()` masks out all elements for which Boolean is `false`

    ```
    // Assign only elements for which mask is true
    stdx::simd_select(x < 0, x) = -x;
    ```

- All operations on masks are specialized for single `bool` values as well

SIMD Mandelbrot

Aside: Using SIMD with Algorithms

- Normal (non-SIMD) iteration over a given array:

```
std::vector<float> data = {...};
for (size_t i = 0; i != data.size(); ++i)
{
    do_something(data[i]);
}
```

```
void do_something(float t)
{
    // do something with t
}
```

- Equivalent algorithm use:

```
std::for_each(data.begin(), data.end(), do_something);
```

- Equivalent parallel algorithm use:

```
std::for_each(std::execution::par, data.begin(), data.end(), do_something);
```

Aside: Using SIMD with Algorithms

- Normal (SIMD) iteration over a given array:

```
using V = stdx::simd<float>;
size_t i = 0;

// handle vectorizable elements
for (/**/; i + V::size() <= N; i += V::size())
{
    // loads V::size() elements
    stdx::simd x(&data[i]);
    do_something(x);
}

// handle epilogue elements (if any)
for (/**/; i < N; ++i)
{
    do_something(data[i]);
}
```

```
void do_something(stdx::simd<float> t)
{
    // do something with t
}

void do_something(float t)
{
    // do something with t
}
```

```
template <typename T>
void do_something(T t)
{
    // do something with t
}
```

Aside: Using SIMD with Algorithms

- Equivalent algorithm use (only vectorization):

```
hpx::for_each(hpx::execution::simd,
    data.begin(), data.end(), do_something);
```

- Equivalent algorithm use (also parallelize):

```
hpx::for_each(hpx::execution::par_simd,
    data.begin(), data.end(), do_something);
```

```
auto do_something = [](auto t)
{
    // do something with t
}
```

SIMD Mandelbrot

```
stdx::simd<size_t> mandelbrot(stdx::simd<std::complex<float>> z,
    stdx::simd<std::complex<float>> const c, stdx::simd<size_t> count)
{
    size_t const max_iterations = 100;
    auto msk = abs(z) > 2.0f;          // evaluates to stdx::simd_mask
    for (size_t k = 0; k < max_iterations; ++k) {
        z = z * z + c;
        // assign iteration count to newly found elements > 2
        auto curr_msk = abs(z) > 2.0f;
        stdx::simd_select(msk ^ curr_msk, count) = k;
        if (std::all_of(curr_msk))
            return count;              // break if all elements diverge
        msk = curr_msk;
    }
    return count;
}
```

SIMD Mandelbrot

```
size_t mandelbrot(std::complex<float> z,
    std::complex<float> const c, size_t count)
{
    size_t const max_iterations = 100;
    auto msk = abs(z) > 2.0f;          // evaluates to bool
    for (size_t k = 0; k < max_iterations; ++k) {
        z = z * z + c;
        // assign iteration count to newly found elements > 2
        auto curr_msk = abs(z) > 2.0f;
        stdx::simd_select(msk ^ curr_msk, count) = k;
        if (std::all_of(curr_msk))
            return count;               // break if all elements diverge
        msk = curr_msk;
    }
    return count;
}
```

SIMD Mandelbrot

```
template <typename T, typename Count>
Count mandelbrot(T z, T const c, Count count) {
    size_t const max_iterations = 100;
    auto msk = abs(z) > 2.0f;
    for (size_t k = 0; k < max_iterations; ++k) {
        z = z * z + c;
        // assign iteration count to newly found elements > 2
        auto curr_msk = abs(z) > 2.0f;
        stdx::simd_select(msk ^ curr_msk, count) = k;
        if (std::all_of(curr_msk))
            return count;   // break if all elements diverge
        msk = curr_msk;
    }
    return count;
}
```

SIMD Mandelbrot

- Slightly more complicated
  - The elements of the `simd` variable may require different number of iterations
  - Break out of loop once all elements have diverged
    - Return iteration count for each element
  - Otherwise return -1 for elements that do not diverge
- This function is fully generic:

```
size_t m = mandelbrot(
    std::complex<float>(0.0, 0.0),
    std::complex<float>(c), size_t(-1));

stdx::simd<size_t> m = mandelbrot(
    stdx::simd<std::complex<float>>(0.0, 0.0),
    stdx::simd<std::complex<float>>(c), stdx::simd<size_t>(-1));
```

SIMD Mandelbrot

```
template <typename T, typename Const, typename Count>
Count mandelbrot(T z, Const const c, Count count) {
    size_t const max_iterations = 100;
    auto msk = abs(z) > 2.0f;
    fixed_point(
        [c](T z) { return z = z * z + c; },
        [&](T, T z) {
            auto curr_msk = abs(z) > 2.0f;
            stdx::simd_select(msk ^ curr_msk, count) = k;
            if (std::all_of(curr_msk))
                return true;            // break if all elements diverge
            msk = curr_msk;
            return false;
        },
        max_iterations, z);
    return count;
}
```

SIMD Execution Policy

```
hpx::for_each(hpx::execution::par,
    counting_iterator(0), counting_iterator(size_y),
    [&](size_t pixel_y) {
        std::simd<float> imag = scale(pixel_y, 0, size_y, -2.0, 2.0);
        hpx::experimental::for_loop(hpx::execution::simd,
            0, size_x,
            [=, &mandelbrot_img](auto pixels_x) {
                std::simd<float> real = scale(pixels_x, 0, size_x, -2.0, 2.0);
                auto values = mandelbrot(std::simd<std::complex>(),
                    std::simd<std::complex>({real, imag}),
                    std::simd<size_t>());
                write_pixels(mandelbrot_img, pixels_x, pixel_y, values);
            });
        });
```

Requires: P2663: Interleaved complex values support in std::simd

SIMD Execution Policy

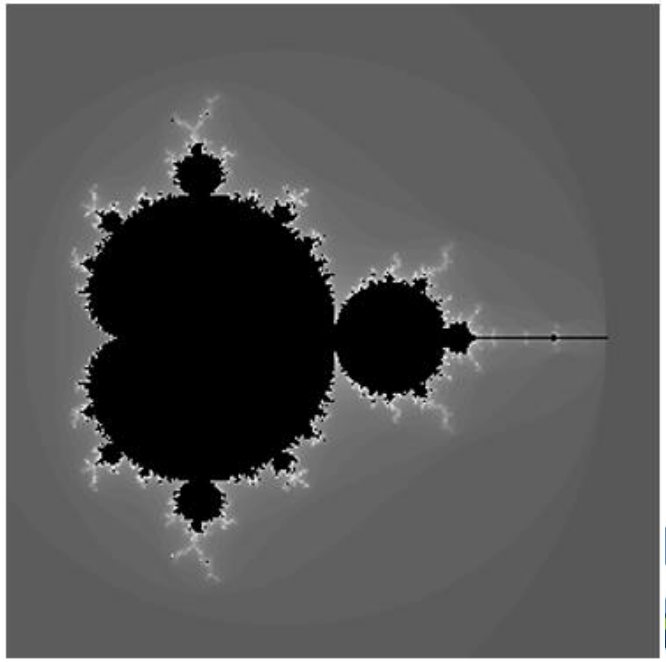
```
hpx::experimental::for_loop(hpx::execution::par,
    0, size_y,
    [&](size_t pixel_y) {
        std::simd<float> imag = scale(pixel_y, 0, size_y, -2.0, 2.0);
        hpx::experimental::for_loop(hpx::execution::simd,
            0, size_x,
            [=, &mandelbrot_img](auto pixels_x) {
                std::simd<float> real = scale(pixels_x, 0, size_x, -2.0, 2.0);
                auto values = mandelbrot(std::simd<std::complex>(),
                    std::simd<std::complex>({real, imag}),
                    std::simd<size_t>());
                write_pixels(mandelbrot_img, pixels_x, pixel_y, values);
            });
        });
```

Requires: P2663: Interleaved complex values support in std::simd

Area of Mandelbrot Set

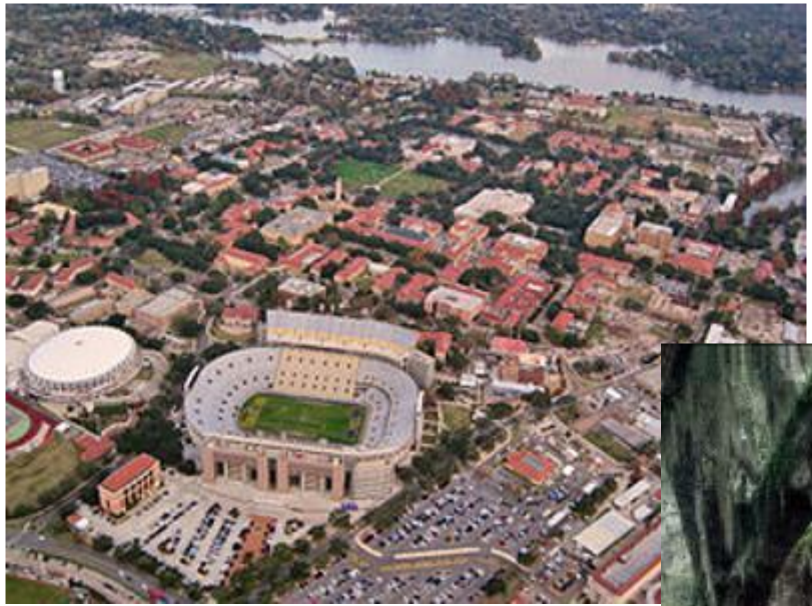
The Mandelbrot Set: The Area

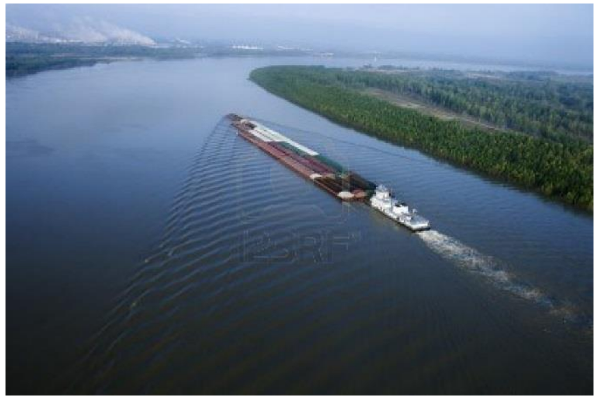
- The area of Mandelbrot set has been theoretically approximated: $1.506591856 \pm 2.54\times10^{-8}$
  - https://mrob.com/pub/muency/areaofthemandelbrotset.html
  - https://github.com/MaartenStork/Mandelbrot_Set_Approximation/blob/main/Report.pdf
- Computational approximation possible:
  - Use Monte-Carlo method for this
  - Repeat until satisfied:
    - Randomly select point in complex plane
    - Count if point is part of Mandelbrot Set
  - Compute ration of successful hits and overall attempts



Exercise

- Compute area of the Mandelbrot set using Monte-Carlo methods
- Parallelelize the code that approximates the area of the Mandelbrot Set
  - Use HPX for_loop with reduction helper
- How many iterations did you need for decent approximation?
- Plot the approximated value and the estimation error over the number of attempts


LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP