PHI

A Modern C++ Library for Parallel Pattern Composition

Santiago Veigas Ramírez

sveigas@inf.uc3m.es

Daniel Martínez Davies

J. Daniel García Sánchez

uc3m Universidad Carlos III de Madrid

TABLE OF CONTENTS

Introduction

- Programming languages evolve, so does the way to express code.
- Pattern-based programming improves readability and maintainability.
- Ranges enable more expressive, composable, and safer code.
- Transition from iterators to ranges.
- Parallel code can benefit from similar abstractions.

C++ 17
Parallel Algorithms

C++ 20
Ranges & Views

C++ 23
User-defined
Range Adaptors

C++ 26
Reflection (๑•̀ワ•́)و
Execution
Parallel Ranges

TABLE OF CONTENTS

C++ 20 Ranges | Background

- High-level abstraction that supersedes iterators.
  - Introduces the concepts of ranges & views.
- Replaces the use of iterator-pairs with a pipe-like syntax.
- Generally lazily evaluated.

```cpp
// View Composition
auto && squares = std::views::iota(1)                    // Infinite Range [1 ... N]
| std::views::transform([] (int x) { return x * x; })   // Square elements
| std::views::filter([] (int x) { return x > 100; })    // Keep elements > 100
| std::views::take(10)                                   // Keep 10 elements
;
// View Evaluation (Print Elements)
for (auto element: squares) std::print("{} ", element);
// Another Evaluation (Print Average)
std::print("[{}]", std::ranges::fold_left(squares, 0.0 ,std::plus{}) / 10.0);
```

https://godbolt.org/z/ehvIEfsze

Parallel Patterns | Background

- Programming patterns are widely recognized as a best practice.
- They define a clear mapping between input and output data, using a transformation function to express the computation to be done in parallel
- Parallel control patterns (data parallel patterns)
  - Map, reduce, stencil
- Parallel data management patterns
  - Pipeline, pack/unpack, scatter/gather

Syntax | Background

- Commonly found in OpenMP.
- Primarily based on pragma directives.
- Requires minimal code changes.
- Limited type support.
- Obscure error messages.

Macros

```c
void openmp(float a[], float b[], int n) {
  int i, j;
  #pragma omp parallel shared(a,b,n)
  {
   #pragma omp for schedule(dynamic,1) private (i,j) nowait
    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++)
          b[j + n*i] = (a[j + n*i] + a[j + n*(i-1)]) / 2.0;
  }
}
```

Syntax | Background

- No method chaining.
- Often stateless and reusable.
- Could be composable but not "fluent".
- Most common among parallel frameworks and libraries.

Free Functions

```cpp
// Intel TBB
tbb::parallel_for(0, static_cast<int>(v1.size()), [&](int i) {
  v1[i] = i * i;
});
```

```cpp
// C++17 Parallel Algorithms
std::for_each(std::execution::parallel_policy{},
  v1.begin(), v1.end(),
  [] (int &x) { x = x*2; }
);
```

```cpp
// GrPPI
grppi::map(grppi:: parallel_execution_native{}, v1, v1,
  [](int &x) { x = x * 2; }
);
```

Syntax | Background

- Declarative, functional style.
- Chainable operations.
- Returns *this or new object for chaining.
- Similar to range-like pipes.

Fluent / Chaining Style

```cpp
// RxCPP
auto values = rxcpp::observable<>::range(1, 10)
  .map([](int v) { return v * 2; })
  .filter([](int v) { return v % 2 == 0; })
  .reduce(0, [](int acc, int v) { return acc + v; });
```

Syntax | Background

- Heterogenous frameworks
  - SYCL
  - CUDA
  - OpenCL
- Work-items and work-groups
- Manual index calculation
- Direct memory access

Kernel

```
// SYCL 2020
q.submit([&](sycl::handler& h) {
  auto in  = in_buf.get_access<sycl::access::mode::read>(h);
  auto out = out_buf.get_access<sycl::access::mode::write>(h);
  h.parallel_for(sycl::range<1>{N}, [=](sycl::id<1> i) {
    out[i] = in[i] * in[i];
  });
});
```

```
// CUDA
__global__ void square_kernel(int* input, int* output, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    output[i] = input[i] * input[i];
}
```

TABLE OF CONTENTS

Interface | Design

- PHI builds on the foundations and concepts of GrPPI.
- PHI's core abstractions:
    - Range adaptors.
    - Data-parallel patterns.
    - Execution backends.
- Pattern composition & execution should be decoupled.
- Parallel Patterns are modeled after Range Adaptors.
- Avoid explicitly composed patterns like map-reduce.
- Type constraints.

GitHub

https://github.com/arcosuc3m/grppi

Range Adaptors | Design

- Range Adaptors are Customization Point Objects (CPOs) that transform a range into another range-like object.
- They create deeply nested types.

```cpp
// Using pipe syntax
auto && view = range | adaptor_1 | adaptor_2;
// Equivalent without RACOs
auto && view = adaptor_2(adaptor_1(range));
```

```cpp
// std::ranges::views::__adaptor::_Partial<std::views::_Transform, lambda(int)>
auto && T = std::views::transform([] (int x) {return x;});

// std::ranges::transform_view<std::ranges::iota_view<int, std::unreachable_sentinel_t>, lambda(int)>
auto && I = std::views::iota(1) | T;
```

Range Adaptor Closures | Design

- RACOs are unary function objects that enables the piping mechanism.
  - When applied to a range: the output is a view.
  - When applied to a RACO: the output is a closure object.

```cpp
struct identity_fn : std::ranges::range_adaptor_closure<identity_fn> {
  template<std::ranges::viewable_range Rng>
  constexpr auto operator()(Rng && range) const {
    return std::forward<Rng>(range);
  }
};
inline constexpr auto identity = identity_fn{};

// Usage Example
int main() {
  for (auto x : std::vector{1,2,3,4} | identity)
    std::print("{} ", x); // Prints 1 2 3 4
}
```

https://godbolt.org/z/835GhMaE8

Map Adaptors | Design

uc3m Universidad Carlos III de Madrid

- Modeled after Unary functions.
- N-ary interface substituted by zip views over data.
- Unary Transformer: **U res = op(x)**
- Transformer function must be pure.

```
// Illustrative Interface - Unary Map Transformer
auto transformer = [ ] (T element) -> U
    { return /* Transformation Op */ ; };

// Usage Example
auto pipeline = input_range<T>
    | phi::map(transformer)
    | phi::map([ ] (U element) { return 42 * element; })
    ;
```

Reduce Adaptors | Design

- Modeled after a combiner function & identity value.
- Combiner: **T res = cmb(T x, U y).**
- Combiner function should be pure & associative.
- Special type of adaptor → "Terminal Operation / Range"

```
// Illustrative Interface - Reduce Combiner
auto combiner = [ ] (T element_1, U element_2) → T
    { return /* Combination Op */; };

// Usage Example
auto pipeline = input_range<T>
    | phi::map([ ] (T e) → T { return e * e; } )
    | phi::reduce([ ] (T e1, U e2) { return e1 + e2; } , 0) // Identity Value
//  | phi::reduce(std::plus{}, 0)                           // Equivalent Expression
    ;
```

Type Requirements | Design

- Performance impact & implementation challenges
  - Irregularly structured data in memory.
  - Data that cannot be partitioned into parallel units.
- C++ Proposal P3179 authors' identify random access iterators as a common requirement across many existing parallel execution models.
- Ranges must be bounded in size.
  - std::ranges::iota_view produces unbounded sequences.

TABLE OF CONTENTS

Mandelbrot | Evaluation

- Embarrassingly parallel escape-time algorithm.
- Well-suited for map-pattern decomposition.

- The algorithm has these stages:
  - Initialization of a grid of data points.
  - Mapping each grid index to a corresponding coordinate in the complex plane.
  - Running the escape-time iteration for each complex point.

```cpp
// Precompute the change in real and imaginary components (x_step, y_step)
double const x_step = (plane.x_max - plane.x_min) / image_width;
double const y_step = (plane.y_max - plane.y_min) / image_height;

// Initialize the matrix (std::vector<unsigned>) from 0 to N
std::ranges::iota(matrix, 0);

// Execution loop
for (unsigned const n : matrix) {
  // Convert 1D index to 2D index (i, j)
  unsigned x = n % matrix.width;
  unsigned y = n / matrix.width;

  // Map index to complex plane coordinates (x, y)
  double real = plane.x_min + (x * x_step);
  double imag = plane.y_min + (y * y_step);

  // Compute escape time for the complex point
  std::complex<double> c(real, imag);
  std::complex<double> z(0.0, 0.0);
  unsigned iterations = 0;
  while (std::norm(z) <= escape_value * 2 && iterations < max_iterations) {
    z = std::pow(z, 2) + c;
    ++iterations;
  }
  output[n] = iterations; // Assume that output is correctly sized
}
```

Mandelbrot | Evaluation

Traditional Algorithm

- All stages of the computation are embedded directly in the inner loop.
- Executed eagerly at each iteration
- Algorithm and implementation details are coupled.
- More complex algorithms could introduce errors with indexed array accesses.

```cpp
// Precompute the change in real and imaginary components (x_step, y_step)
// x_min, x_max, y_min, and y_max represent a window in the complex plane
double const x_step{(plane.x_max - plane.x_min) / image_width};
double const y_step{(plane.y_max - plane.y_min) / image_height};

// Initialize the matrix (std::vector<unsigned>) from 0 to N
std::ranges::iota(matrix, 0);

// Transform iota value (n) to an idx value (i, j) → 1D index to 2D index
auto iota_to_idx = [&] (unsigned const value) → coordinates<unsigned> {
  return {.x = value % matrix.width, .y = value / matrix.width};
};

// Transform the idx pair to a coordinate pair belonging to the Mandelbrot set
auto idx_to_xy = [=] (coordinates<unsigned> const & ij)
    → coordinates<double> {
  return {.x = x_min + (ij.x * x_step), .y = y_min + (ij.y * y_step)};
};

// Compute escape time for the complex point
auto xy_to_escape = [=] (coordinates<double> const & xy) → unsigned {
  unsigned iterations{0};
  std::complex const c{xy.x, xy.y};
  std::complex<double> z{0, 0};
  while (std::norm(z) ≤ escape_value * 2 && iterations < max_iterations) {
    z = pow(z, 2) + c;
    ++iterations;
  }
  return iterations;
};
```

Mandelbrot | Evaluation

Lambda Algorithmic Skeleton

- Loop has been replaced with lambdas representing data transformations.
  - A single lambda would've sufficed.
- Less intuitive than the iterative version, but easier to compose, reuse, and abstract.
- None of the lambdas are eagerly executed.

Mandelbrot | Evaluation

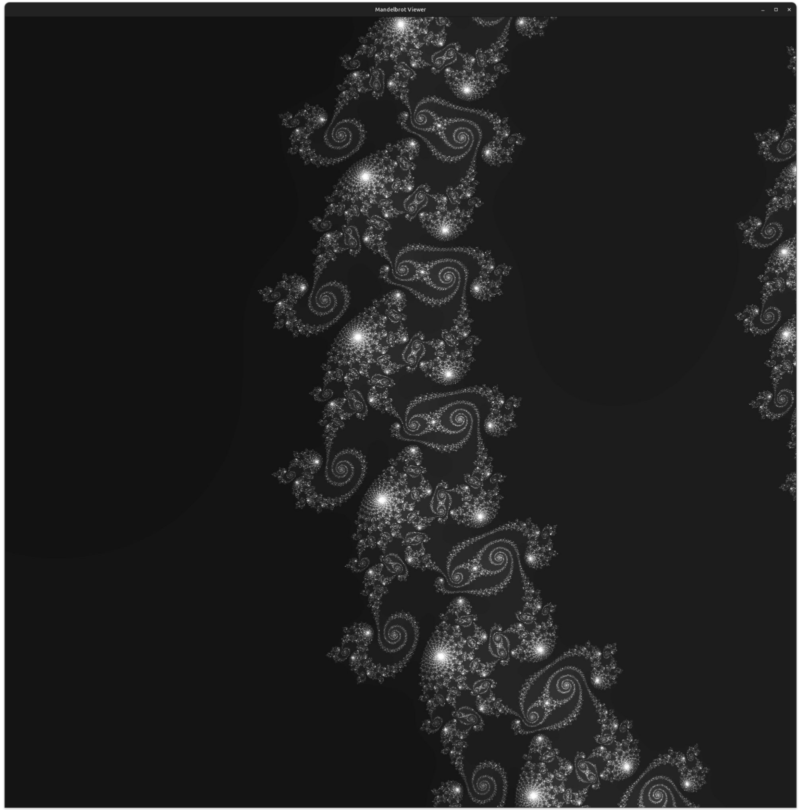
- Each lambda can now be composed into declarative pipelines.
- Lambdas can be re-used with minimal assumptions about the underlying data range.
- The pipeline can be executed iteratively as a range-loop; or passed to an offloading backend.

```
// Pattern composition
auto && pipeline =
    matrix
    | phi::map(iota_to_idx)
    | phi::map(idx_to_xy)
    | phi::map(xy_to_escape)
    ;

// Pipeline execution
phi::execute(pipeline, output);
```

Mandelbrot | Evaluation

Machine Configuration	
CPU (X86_64)	AMD Ryzen 9 5950X
Cores	16 (2 threads per core)
Base - Max Frequency	2200 MHz - 5083 MHz
L1 Cache	512 KiB + 512 KiB
L2 Cache	8 MiB
L3 Cache	64 MiB (2x)
Compilation Configuration	
Compiler	GCC 14.2.0
Flags	-O3 -DNDEBUG
Mandelbrot Configuration	
Position	-0.761574 -0.0847596 3125
Iterations	10,000
Escape Value	2.0



Mandelbrot | Evaluation

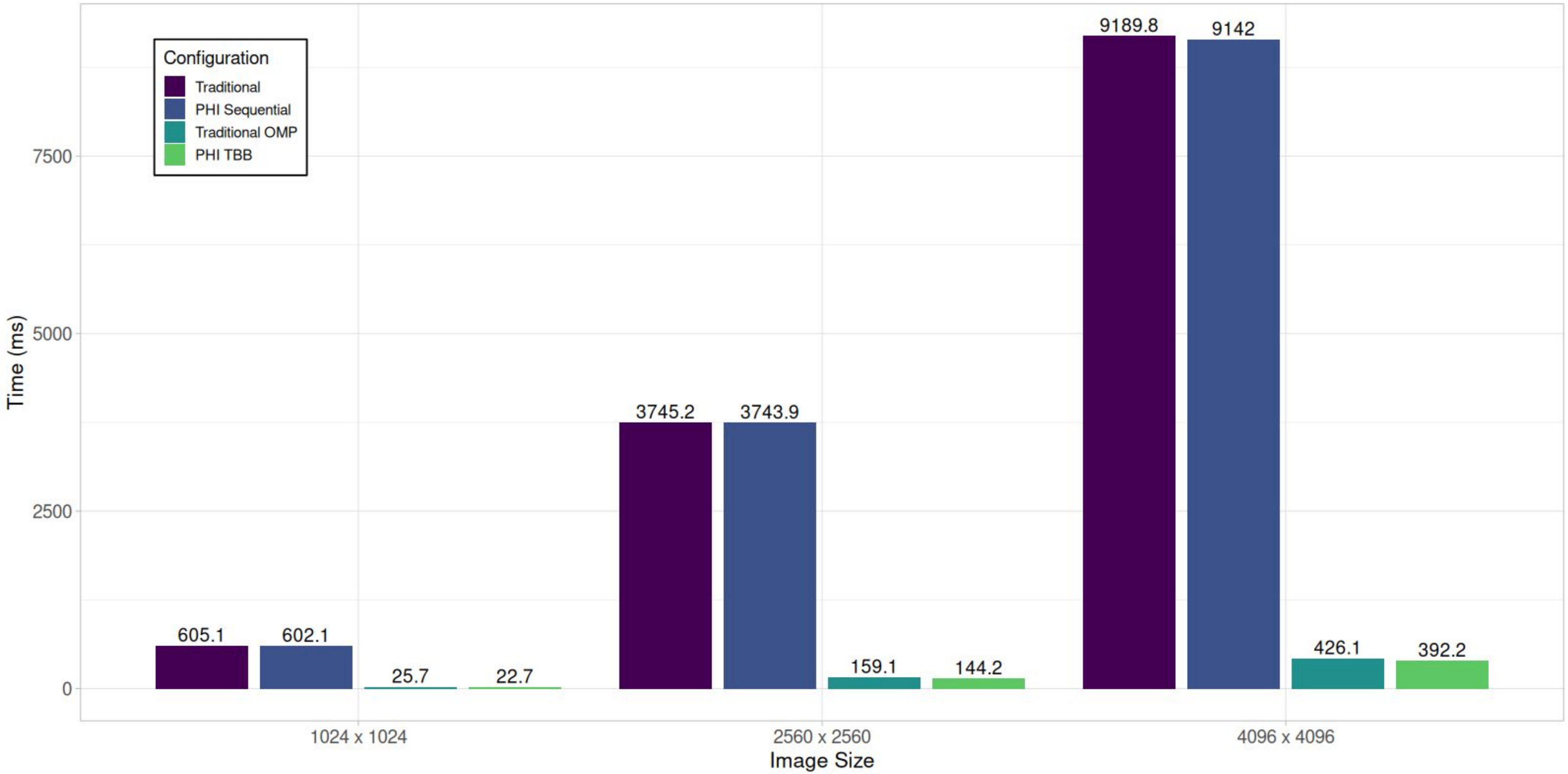


TABLE OF CONTENTS

Conclusion

- PHI leverages more idiomatic C++ range-like interfaces.
- Range adaptors and closure objects are the foundation of function composition for lazy evaluation.
- PHI is in early stages of development. Improvements need to be made.
    - More pattern adaptors: ***stencil - scan - filter - zip***.
    - Support for in-place data. Requires careful study.
    - Improved support for backend integration, and more backends.
        - Deeply nested types are difficult to work with. Study a better representation.
    - Non-linear chains of transformation.
    - Additional and more complex use-cases.

PHI

A Modern C++ Library for Parallel Pattern Composition

Santiago Veigas Ramírez
sveigas@inf.uc3m.es

Daniel Martínez Davies
J. Daniel García Sánchez

uc3m Universidad Carlos III de Madrid