An introduction to **GrPPI**

Generic Reusable Parallel Patterns Interface

J. Daniel Garcia

ARCOS Group
University Carlos III of Madrid
Spain

February 2018

Warning

ARCOS@uc3m

- **UC3M**: A young international research oriented university.
- **ARCOS**: An applied research group.
  - Lines: High Performance Computing, Big data, Cyberphysical Systems, and **Programming models for application improvement**.
- **Programming Models for Application Improvement**:
  - Provide programming tools for **improving**:
    - Performance.
    - Energy efficiency.
    - Maintainability.
    - Correctness.
- **Standardization**:
  - **ISO/IEC JTC/SC22/WG21**. ISO C++ standards committe.

Acknowledgements

- The GrPPI library has been partially supported by:
  - Project ICT 644235 **"REPHRASE: REfactoring Parallel Heterogeneous Resource-aware Applications"** funded by the European Commission through H2020 program (2015-2018).
  - Project TIN2016-79673-P **"Towards Unification of HPC and Big Data Paradigms"** funded by the Spanish Ministry of Economy and Competitiveness (2016-2019).

GrPPI team

- **Main team**
  - J. Daniel Garcia (UC3M, lead).
  - David del Río (UC3M).
  - Manuel F. Dolz (UC3M).
  - Javier Fernández (UC3M).
  - Javier Garcia Blas (UC3M).
- **Cooperation**
  - Plácido Fernández (UC3M-CERN).
  - Marco Danelutto (Univ. Pisa)
  - Massimo Torquati (Univ. Pisa)
  - Marco Aldinucci (Univ. Torino)
  - Fabio Tordini (Univ. Torino)
  - ...

Sequential Programming versus Parallel Programming

- **Sequential programming**
  - Well-known set of *control-structures* embedded in programming languages.
  - Control structures inherently sequential.

Sequential Programming versus Parallel Programming

- **Sequential programming**
  - Well-known set of *control-structures* embedded in programming languages.
  - Control structures inherently sequential.
- **Traditional Parallel programming**
  - Constructs *adapting* sequential control structures to the parallel world (e.g. *parallel-for*).

Sequential Programming versus Parallel Programming

- **Sequential programming**
  - Well-known set of *control-structures* embedded in programming languages.
  - Control structures inherently sequential.
- **Traditional Parallel programming**
  - Constructs *adapting* sequential control structures to the parallel world (e.g. *parallel-for*).
- But wait!
  - What if we had constructs that could be both sequential and parallel?

Software design

There are two ways of constructing a software design:

Software design

There are two ways of constructing a software design:

One way is

Software design

There are two ways of constructing a software design:

One way is
to make it ***so simple*** *that there are* ***obviously no deficiencies****,*

Software design

There are two ways of constructing a software design:

One way is
to make it ***so simple*** *that there are* ***obviously no deficiencies****,*

and the other way is

Software design

There are two ways of constructing a software design:

One way is
to make it ***so simple*** *that there are* ***obviously no deficiencies****,*

and the other way is
to make it ***so complicated*** *that there are* ***no obvious deficiencies****.*

Software design

There are two ways of constructing a software design:

One way is
to make it ***so simple*** *that there are* ***obviously no deficiencies****,*

and the other way is
to make it ***so complicated*** *that there are* ***no obvious deficiencies****.*

The ***first method*** *is* ***far more difficult****.*

C.A.R Hoare

Adding two vectors

Traditional way

```
using numvec = std::vector<double>;

numvec add(const numvec & v1, const numvec & v2) {
  numvec res;
  res.reserve(v1.size()) ; // Asume equal sizes
  for (int i =0; i <v1.size(); ++i) {
    res.push_back(v1[i]+v2[i]) ;
  }
  return res;
}
```

Adding two vectors

Traditional way

```
using numvec = std::vector<double>;

numvec add(const numvec & v1, const numvec & v2) {
  numvec res;
  res.reserve(v1.size()) ; // Asume equal sizes
  for (int i =0; i <v1.size(); ++i) {
    res.push_back(v1[i]+v2[i]) ;
  }
  return res;
}
```

- Adds additional constraints.
  - Traversing in-order.
- Potential mistakes.
  - **i<v1.size()** versus **i<=v1.size()**.

Adding two vectors

The STL way

```
using numvec = std::vector<double>;

numvec add(const numvec & v1, const numvec & v2) {
  numvec res;
  res.reserve(v1.size () ) ; // Asume equal sizes
  std::transform(v1.begin(), v1.end(), v2.begin() ,
      std::back_inserter(res),
      []( double x, double y) { return x+y; }
  );
  return res;
}
```

Adding two vectors

The STL way

```
using numvec = std::vector<double>;

numvec add(const numvec & v1, const numvec & v2) {
  numvec res;
  res.reserve(v1.size()); // Asume equal sizes
  std::transform(v1.begin(), v1.end(), v2.begin(),
      std::back_inserter(res),
      [](double x, double y) { return x+y; }
  );
  return res;
}
```

- Does not add additional constraints (ordering).
- Less error prone.

A brief history of patterns

- From building and architecture (Cristopher Alexander):
  - **1977**: A Pattern Language: Towns, Buildings, Construction.
  - **1979**: The timeless way of buildings.
- To software design (Gamma et al.):
  - **1993**: Design Patterns: abstraction and reuse of object oriented design. ECOOP.
  - **1995**: Design Patterns. Elements of Reusable Object-Oriented Software.
- To parallel programming (McCool, Reinders, Robinson):
  - **2012**: Structured Parallel Programming: Patterns for Efficient Computation.

GrPPI ideals

- Applications should be expressed **independently** of the **execution model**.

GrPPI ideals

- Applications should be expressed **independently** of the **execution model**.
- **Computations** should be expressed in terms of **structured composable patterns**.

GrPPI ideals

- Applications should be expressed **independently** of the **execution model**.
- **Computations** should be expressed in terms of **structured composable patterns**.
- **Multiple back-ends** should be offered with **simple switching** mechanisms.

GrPPI ideals

- Applications should be expressed **independently** of the **execution model**.
- **Computations** should be expressed in terms of **structured composable patterns**.
- **Multiple back-ends** should be offered with **simple switching** mechanisms.
- Interface should **integrate** seamlessly with **modern C++** and its standard library.

GrPPI ideals

- Applications should be expressed **independently** of the **execution model**.
- **Computations** should be expressed in terms of **structured composable patterns**.
- **Multiple back-ends** should be offered with **simple switching** mechanisms.
- Interface should **integrate** seamlessly with **modern C++** and its standard library.
- Applications should be able to **take advantage** of **modern C++** language features.

GrPPI

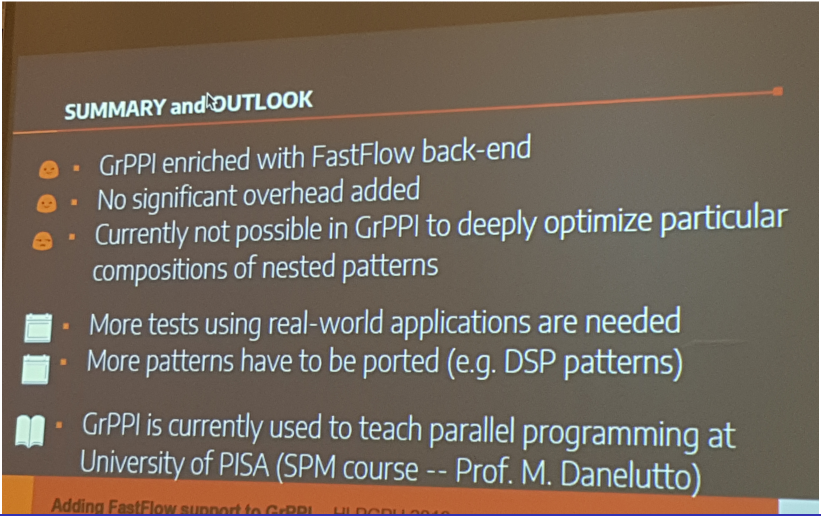
https://github.com/arcosuc3m/grppi

GrPPI

https://github.com/arcosuc3m/grppi

- A header only library (might change).
- A set of execution policies.
- A set of type safe generic algorithms.
- Requires **C++14**.
- GNU GPL v3.

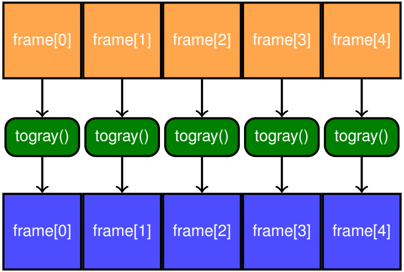
GrPPI as a teching tool



Example: Transforming a sequence

- Given a sequence of **frames** generate a new sequence of frames in grayscale.

```
struct frame { /* ... */ };
frame togray(const frame & f);
```



Transforming a sequence

Traditional explicit loop

```
using frameseq = std::vector<frame>;

frameseq seq_togray(const frameseq & s) {
  frameseq r;
  r.reserve(s.size());

  // Requires processing in-order
  for (const auto & f : s) {
    r.push_back(togray(f));
  }
  return r;
}
```

Transforming a sequence

STL way

```
using frameseq = std::vector<frame>;

frameseq seq_togray(const frameseq & s) {
  frameseq r;
  r.reserve(s.size());

  std::transform(s.begin(), s.end(), std::back_inserter(r), togray);

  return r;
}
```

Transforming a sequence

Parallel STL way (C++17)

```
using frameseq = std::vector<frame>;

frameseq seq_togray(const frameseq & s) {
  frameseq r(s.size());

  // No execution order assumed
  std::transform(std::par, s.begin(), s.end(), std::back_inserter(r), togray);

  return r;
}

int main() {
```

Transforming a sequence

GrPPI (map pattern)

```
using frameseq = std::vector<frame>;

frameseq seq_togray(const frameseq & s) {
  frameseq r(s.size());

  grppi::sequential_execution seq;
  grppi::map(seq, s.begin(), s.end(), r.begin(), togray);

  return r;
}
```

Transforming a sequence

GrPPI + lambda

```
using frameseq = std::vector<frame>;

frameseq seq_togray(const frameseq & s) {
  frameseq r(s.size());

  grppi::sequential_execution seq;
  grppi::map(seq, s.begin(), s.end(), r.begin(),
    [](const frame & f) { return filter(f,64); });

  return r;
}
```

Execution types

- Execution model is encapsulated in execution types.
  - Always provided as first argument to patterns.
- Current concrete execution types:
  - **Sequential**: **sequential_execution**.
  - **ISO C++ Threads**: **parallel_execution_native**.
  - **OpenMP**: **parallel_execution_omp**.
  - **Intel TBB**: **parallel_execution_tbb**.
  - **FastFlow**: **parallel_execution_ff**.
- Run-time polymorphic wrapper through type erasure:
  - **dynamic_execution**.

Execution model properties

- Some execution types allow finer configuration.
  - Example: Concurrency degree.

Execution model properties

- Some execution types allow finer configuration.
  - Example: Concurrency degree.

- Interface:

```
ex.set_concurrency_degree(4);
int n = ex.concurrency_degree();
```

Execution model properties

- Some execution types allow finer configuration.
  - Example: Concurrency degree.

- Interface:

```
ex.set_concurrency_degree(4);
int n = ex.concurrency_degree();
```

- **Default values**:
  - **Sequential** ⇒ **1**.
  - **Native** ⇒ **std::thread::hardware_concurrency()**.
  - **OpenMP** ⇒ **omp_get_num_threads()**.

Upcoming execution types

- **parallel_execution_cuda**
  - Support for **CUDA** devices through **Thrust**.
- **parallel_execution_ocl**
  - Support for **OpenCL** devices through **SYCL**.
- **parallel_execution_mpi**
  - Support for **MPI**.

A classification

- **Data patterns**: Express computations over a data set.
  - **map**, **reduce**, **map/reduce**, **stencil**.

A classification

- **Data patterns**: Express computations over a data set.
  - **map**, **reduce**, **map/reduce**, **stencil**.
- **Task patterns**: Express task composition.
  - **divide/conquer**.

A classification

- **Data patterns**: Express computations over a data set.
  - **map**, **reduce**, **map/reduce**, **stencil**.
- **Task patterns**: Express task composition.
  - **divide/conquer**.
- **Streaming patterns**: Express computations over a (possibly unbounded) data stream.
  - **pipeline**.
  - Specialized stages: **farm**, **filter**, **reduction**, **iteration**.

3 Patterns in GrPPI

- Controlling execution
- Patterns overview
- **Data patterns**
- Task Patterns
- Streaming patterns

Patterns on data sets

- A **data pattern** performs an operation on one or more data sets that are already in memory.

- **Input**:
  - One or more data sets.
  - Operations.

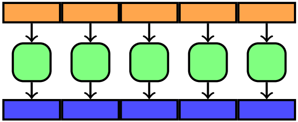
- **Output**:
  - A data set (**map**, **stencil**).
  - A single value (**reduce**, **map/reduce**).

Maps on data sequences

- A **map** pattern applies an operation to every element in a a data set, generating a new data set.

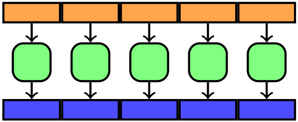
Maps on data sequences

- A **map** pattern applies an operation to every element in a a data set, generating a new data set.
- **Unidimensional**:

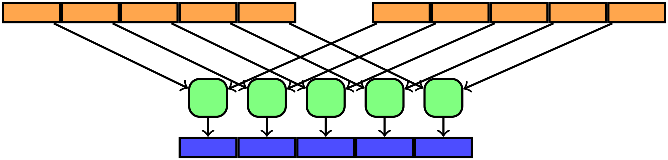


Maps on data sequences

- A **map** pattern applies an operation to every element in a a data set, generating a new data set.
- **Unidimensional**:



- **Multidimensional**:



Single sequences mapping

Double all elements in vector sequentially

```
std::vector<double> double_elements(const std::vector<double> & v)
{
  std::vector<double> res(v.size());
  grppi::sequential_execution seq;

  grppi::map(seq, v.begin(), v.end(), res.begin(),
    []( double x) { return 2*x; });

  return res;
}
```

Single sequences mapping

Double all elements in vector with OpenMP

```
std::vector<double> double_elements(const std::vector<double> & v)
{
  std::vector<double> res(v.size());
  grppi::parallel_execution_omp omp;

  grppi::map(omp, v.begin(), v.end(), res.begin(),
    [](double x) { return 2*x; });

  return res;
}
```

Multiple sequences mapping

Add two vectors

```
template <typename Execution>
std::vector<double> add_vectors(const Execution & ex,
                                const std::vector<double> & v1,
                                const std::vector<double> & v2)
{
  auto size = std::min(v1.size(), v2.size());
  std::vector<double> res(size);

  grppi::map(ex, std::make_tuple(v1.begin(), v2.begin), v1.end(),
    res.begin(),
     []( double x, double y) { return x+y; },
  );

  return res;
}
```

Multiple sequences mapping

Add three vectors

```
template <typename Execution>
std :: vector<double> add_vectors(const Execution & ex,
                                  const std::vector<double> & v1,
                                  const std::vector<double> & v2,
                                  const std::vector<double> & v3)
{
  auto size = std :: min(v1.size() , v2.size () ) ;
  std :: vector<double> res(size);

  grppi :: map(ex, std::make_tuple(v1.begin(), v2.begin(), v3.begin()) , v1.end(),
    res.begin() ,
    []( double x, double y, double z) { return x+y+z; },
  ) ;

  return res;
}
```

Heterogeneous mapping

- The result can be from a different type.

Complex vector from real and imaginary vectors

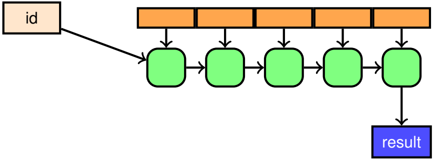
```
template <typename Execution>
std::vector<complex<double>> create_cplx(const Execution & ex,
                                          const std::vector<double> & re,
                                          const std::vector<double> & im)
{
  auto size = std::min(re.size(), im.size());
  std::vector<complex<double>> res(size);

  grppi::map(ex, std::make_tuple(re.begin(), im.begin()), re.end(),
    res.begin(),
    [](double r, double i) -> complex<double> { return {r,i}; }
  );

  return res;
}
```

Reductions on data sequences

- A **reduce** pattern combines all values in a data set using a binary combination operation.



Homogeneous reductions

Add a sequence of values

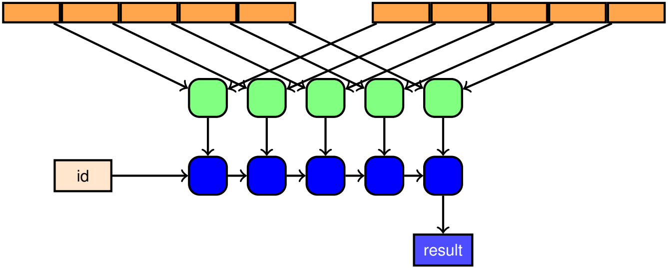
```
template <typename Execution>
double add_sequence(const Execution & ex, const vector<double> & v)
{
  return grppi::reduce(ex, v.begin(), v.end(), 0.0,
    [](double x, double y) { return x+y; });
}
```

Map/reduce pattern

- A **map/reduce** pattern combines a **map** pattern and a **reduce** pattern into a single pattern.
  1. One or more data sets are **mapped** applying a transformation operation.
  2. The results are combined by a **reduction** operation.

- A **map/reduce** could be also expressed by the composition of a **map** and a **reduce**.
  - However, **map/reduce** may potentially fuse both stages, allowing for extra optimizations.

Map/reduce



Single sequence map/reduce

Sum of squares

```
template <typename Execution>
double sum_squares(const Execution & ex, const std::vector<double> & v)
{
  return grppi::map_reduce(ex, v.begin(), v.end(), 0.0,
    [](double x) { return x*x; },
    [](double x, double y) { return x+y; }
  );
}
```

Heterogeneous reductions with map/reduce

Add areas of shapes

```
template <typename Execution>
int add_areas(const Execution & ex, const std::vector<shape> & shapes)
{
  return grppi::map_reduce(ex, shapes.begin(), shapes.end(), 0.0,
    [](const auto & s) { return s.area(); },
    [](double a, double b) { return a+b; }
  );
}
```

- Simpler than heterogeneous reductions.

Map/reduce on two data sets

Scalar product

```
template <typename Execution>
double scalar_product(const Execution & ex,
                      const std::vector<double> & v1,
                      const std::vector<double> & v2)
{
  return grppi::map_reduce(ex, std::make_tuple(begin(v1), begin(v2)), end(v1), 0.0,
    [](double x, double y) { return x*y; },
    [](double x, double y) { return x+y; }
  );
}
```

Cannonical map/reduce

- Given a sequence of words, produce a container where:
  - The key is the word.
  - The value is the number of occurrences of that word.

Cannonical map/reduce

- Given a sequence of words, produce a container where:
  - The key is the word.
  - The value is the number of occurrences of that word.

Word frequencies

```
template <typename Execution>
auto word_freq(const Execution & ex, const std::vector<std::string> & words)
{
```

Cannonical map/reduce

- Given a sequence of words, produce a container where:
  - The key is the word.
  - The value is the number of occurrences of that word.

Word frequencies

```
template <typename Execution>
auto word_freq(const Execution & ex, const std::vector<std::string> & words)
{
  using namespace std;
  using dictionary = std::map<string,int>;
```

Cannonical map/reduce

- Given a sequence of words, produce a container where:
  - The key is the word.
  - The value is the number of occurrences of that word.

Word frequencies

```
template <typename Execution>
auto word_freq(const Execution & ex, const std::vector<std::string> & words)
{
  using namespace std;
  using dictionary = std::map<string,int>;
  return grppi::map_reduce(ex, words.begin(), words.end(), dictionary{},
```

Cannonical map/reduce

- Given a sequence of words, produce a container where:
  - The key is the word.
  - The value is the number of occurrences of that word.

Word frequencies

```
template <typename Execution>
auto word_freq(const Execution & ex, const std::vector<std::string> & words)
{
  using namespace std;
  using dictionary = std::map<string,int>;
  return grppi::map_reduce(ex, words.begin(), words.end(), dictionary{},
    [](string w) -> dictionary { return {w,1}; }
```

Cannonical map/reduce

- Given a sequence of words, produce a container where:
  - The key is the word.
  - The value is the number of occurrences of that word.

Word frequencies

```
template <typename Execution>
auto word_freq(const Execution & ex, const std::vector<std::string> & words)
{
  using namespace std;
  using dictionary = std::map<string,int>;
  return grppi::map_reduce(ex, words.begin(), words.end(), dictionary{},
    [](string w) -> dictionary { return {w,1}; }
    [](dictionary & lhs, const dictionary & rhs) -> dictionary {
      for (auto & entry : rhs) { lhs[entry.first] += entry.second; }
      return lhs;
    });
}
```

Stencil pattern

- A **stencil** pattern applies a transformation to every element in one or multiple data sets, generating a new data set as an output
  - The transformation is function of a data item and its *neighbourhood*.

Single sequence stencil

Neighbour average

```
template <typename Execution>
std::vector<double> neib_avg(const Execution & ex, const std::vector<double> & v)
{
  std::vector<double> res(v.size());
  grppi::stencil(ex, v.begin(), v.end(),
    [](auto it, auto n) {
      return *it + accumulate(begin(n), end(n));
    },
    [&](auto it) {
      vector<double> r;
      if (it!=begin(v)) r.push_back(*prev(it));
      if (distance(it,end(end))>1) r.push_back(*next(it));
      return r;
    });
  return res;
}
```

3 Patterns in GrPPI

- Controlling execution
- Patterns overview
- Data patterns
- **Task Patterns**
- Streaming patterns

Divide/conquer pattern

- A **divide/conquer** pattern splits a problem into two or more independent subproblems until a base case is reached.
  - The base case is solved directly.
  - The results of the subproblems are combined until the final solution of the original problem is obtained.

Divide/conquer pattern

- A **divide/conquer** pattern splits a problem into two or more independent subproblems until a base case is reached.
  - The base case is solved directly.
  - The results of the subproblems are combined until the final solution of the original problem is obtained.

- **Key elements**:
  - **Divider**: Divides a problem in a set of subproblems.
  - **Solver**: Solves and individual subproblem.
  - **Combiner**: Combines two solutions.

A patterned merge/sort

Ranges on vectors

```
struct range {
  range(std::vector<double> & v) : first {v.begin()}, last{v.end()} {}
  auto size() const { return std::distance( first , last ); }
  std::vector<double>::iterator first , last ;
};

std::vector<range> divide(range r) {
  auto mid = r. first + r.size() / 2;
  return { {r. first , mid}, {mid, r. last} };
}
```

A patterned merge/sort

Ranges on vectors

```
template <typename Execution>
void merge_sort(const Execution & ex, std::vector<double> & v)
{
  grppi::divide_conquer(exec, range(v),
    // Divide range in sub−ranges
    []( auto r) −> vector<range> {
      if (1>=r.size()) return {r};
      else return divide(r);
    },
```

A patterned merge/sort

Ranges on vectors

```
template <typename Execution>
void merge_sort(const Execution & ex, std::vector<double> & v)
{
  grppi::divide_conquer(exec, range(v),
    // Divide range in sub−ranges
    [](auto r) −> vector<range> {
      if (1>=r.size()) return {r};
      else return divide(r);
    },
    // A unit range is already ordered
    [](auto x) { return x; },
```

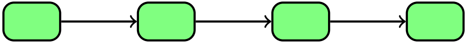
A patterned merge/sort

Ranges on vectors

```
template <typename Execution>
void merge_sort(const Execution & ex, std::vector<double> & v)
{
  grppi::divide_conquer(exec, range(v),
    // Divide range in sub-ranges
    [](auto r) -> vector<range> {
      if (1>=r.size()) return {r};
      else return divide(r);
    },
    // A unit range is already ordered
    [](auto x) { return x; },
    // Merge sorted subranges
    [](auto r1, auto r2) {
      std::inplace_merge(r1.first, r1.last, r2.last);
      return range{r1.first, r2.last};
    });
}
```

Pipeline pattern

- A **pipeline** pattern allows processing a data stream where the computation may be divided in multiple stages.
  - Each stage processes the data item generated in the previous stage and passes the produced result to the next stage.



Standalone pipeline

- A **standalone** **pipeline** is a top-level pipeline.
  - Invoking the pipeline translates into its execution.

Standalone pipeline

- A **standalone** **pipeline** is a top-level pipeline.
  - Invoking the pipeline translates into its execution.

- Given:
  - A **generator** $g : \varnothing \mapsto T_1 \cup \varnothing$
  - A sequence of **transformers** $t_i : T_i \mapsto T_{i+1}$

- For every **non-empty** value generated by g, it evaluates:
  - $t_n(t_{n-1}(\dots t_1(g())))$

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.
- The return value must be any type that:
  - Is copy-constructible or move-constructible.

```
T x = g();
```

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.
- The return value must be any type that:
  - Is copy-constructible or move-constructible.

    ```
    T x = g();
    ```

  - Is contextually convertible to **bool**

    ```
    if (x) { /* ... */ }
    if (!x) { /* ... */ }
    ```

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.
- The return value must be any type that:
  - Is copy-constructible or move-constructible.

    ```
    T x = g();
    ```

  - Is contextually convertible to **bool**

    ```
    if (x) { /* ... */ }
    if (!x) { /* ... */ }
    ```

  - Can be derreferenced

    ```
    auto val = *x;
    ```

Generators

- A generator *g* is any callable C++ entity that:
  - Takes no argument.
  - Returns a value of type *T* that may hold (or not) a value.
  - Null value signals end of stream.
- The return value must be any type that:
  - Is copy-constructible or move-constructible.

    ```
    T x = g();
    ```

  - Is contextually convertible to **bool**

    ```
    if (x) { /* ... */ }
    if (!x) { /* ... */ }
    ```

  - Can be derreferenced

    ```
    auto val = *x;
    ```

- The standard library offers an excellent candidate **std::experimental::optional<T>**.

Simple pipeline

x -> x*x -> 1/x -> print

```
template <typename Execution>
void run_pipe(const Execution & ex, int n)
{
  grppi::pipeline(ex,
    [i=0,max=n] () mutable -> optional<int> {
      if (i<max) return i++;
      else return {};
    },
    [](int x) -> double { return x*x; },
    [](double x) { return 1/x; },
    [](double x) { cout << x << "\n"; }
  );
}
```

Nested pipelines

- **Pipelines** may be **nested**.

- An **inner pipeline**:
  - Does not take an execution policy.
  - All stages are transformers (no generator).
  - The last stage must also produce values.

- The **inner pipeline** uses the same execution policy than the outer pipeline.

Nested pipelines

Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  grppi::parallel_execution_native ex;
```

Nested pipelines

Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  grppi:parallel_execution_native ex;
  grppi::pipeline(ex,
```

Nested pipelines

Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  grppi:parallel_execution_native ex;
  grppi::pipeline(ex,
    [& in_file ]() -> optional<frame> {
      frame f = read_frame(file);
      if (! file ) return {};
      return f;
    },
```

Nested pipelines

Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  grppi:parallel_execution_native ex;
  grppi::pipeline(ex,
    [& in_file ]() -> optional<frame> {
      frame f = read_frame(file);
       if (! file ) return {};
      return f;
    },
    pipeline(
       [](const frame & f) { return filter (f); },
       [](const frame & f) { return gray_scale(f); },
    ),
```

Nested pipelines

Image processing

```
void process(std::istream & in_file , std :: ostream & out_file) {
  grppi:parallel_execution_native ex;
  grppi :: pipeline (ex,
    [& in_file ]()  −> optional<frame> {
      frame f = read_frame(file) ;
      if (! file ) return {};
      return f ;
    },
    pipeline (
      []( const frame & f) { return filter (f) ; },
      []( const frame & f) { return gray_scale(f) ; },
    ),
    [& out_file ]( const frame & f) { write_frame(out_file , f) ; }
  ) ;
}
```

Piecewise pipelines

Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
```

Piecewise pipelines

Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  auto reader = [& in_file ]() -> optional<frame> {
    frame f = read_frame(file);
    if (! file ) return {};
    return f;
  };
```

Piecewise pipelines

Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  auto reader = [& in_file ]() -> optional<frame> {
    frame f = read_frame(file);
    if (! file ) return {};
    return f;
  };
  auto transformer = pipeline(
    [](const frame & f) { return filter (f); },
    [](const frame & f) { return gray_scale(f); },
  );
```

Piecewise pipelines

Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  auto reader = [& in_file ]() −> optional<frame> {
    frame f = read_frame(file);
    if (! file ) return {};
    return f;
  };
  auto transformer = pipeline(
    []( const frame & f) { return filter (f); },
    []( const frame & f) { return gray_scale(f); },
  );
  auto writer = [& out_file ](const frame & f) { write_frame(out_file, f); }
```

Piecewise pipelines

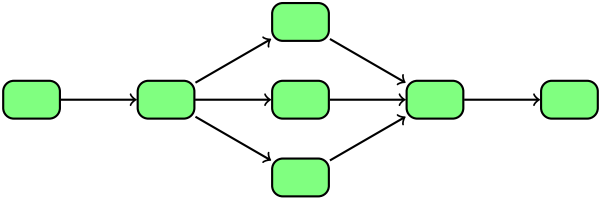
Image processing

```
void process(std::istream & in_file , std::ostream & out_file) {
  auto reader = [& in_file ]() -> optional<frame> {
    frame f = read_frame(file);
    if (! file ) return {};
    return f;
  };
  auto transformer = pipeline(
    []( const frame & f) { return filter (f); },
    []( const frame & f) { return gray_scale(f); },
  );
  auto writer = [& out_file ](const frame & f) { write_frame(out_file, f); }

  grppi:parallel_execution_native ex;
  grppi::pipeline(ex, reader, transformer, writer);
}
```

Farm pattern

- A **farm** is a streaming pattern applicable to a stage in a **pipeline**, providing multiple tasks to process data items from a data stream.
  - A **farm** has an associated **cardinality** which is the number of parallel tasks used to serve the stage.
  - Each task in a **farm** runs a **transformer** for each data item it receives.



Farms in pipelines

Improving a video

```
template <typename Execution>
void run_pipe(const Execution & ex, std::ifstream & filein , std :: ofstream & fileout )
{
  grppi :: pipeline (ex,
    [& filein ] () −> optional<frame> {
      frame f = read_frame(filein ) ;
      if (! filein ) retrun {};
      return f ;
    },
    farm(4, []( const frame & f) { return improve(f); }) ,
    [& fileout ] (const frame & f) { write_frame(f) ; }
  ) ;
}
```

Piecewise farms

Improving a video

```
template <typename Execution>
void run_pipe(const Execution & ex, std::ifstream & filein , std :: ofstream & fileout )
{
  auto reader = [& filein ]  ()  −> optional<frame> {
    frame f  = read_frame(filein) ;
     if  (! filein )  retrun  {};
    return f ;
  };

  auto writer  = [& fileout ]  (const frame & f)  {  write_frame(f) ;  };

  auto improver = farm(4,  []( const frame & f)  {  return improve(f) ;  }) ;

  grppi :: pipeline (ex,  reader,  improver,  writer ) ;
}
```

Ordering

- Signals if pipeline items must be consumed in the same order they were produced.
  - Do they need to be *time-stamped*?

- Default is **ordered**.

- **API**
  - **ex.enable_ordering()**
  - **ex.disable_ordering()**
  - **bool o = ex.is_ordered()**

Queueing properties

- Some policies (**native** and **omp**) use queues to communicate pipeline stages.

- **Properties**:
  - **Queue size**: Buffer size of the queue.
  - **Mode**: *blocking* versus *lock-free*.

- **API**
  - **ex.set_queue_attributes(100, mode::blocking)**

Filter pattern

- A **filter** pattern discards (or keeps) the data items from a data stream based on the outcome of a predicate.
- This pattern can be used only as a stage of a **pipeline**.

Filter pattern

- A **filter** pattern discards (or keeps) the data items from a data stream based on the outcome of a predicate.
- This pattern can be used only as a stage of a **pipeline**.

- **Alternatives**:
  - **Keep**: Only data items satisfying the predicate are sent to the next stage.
  - **Discard**: Only data items **not satisfying** the predicate are sent to the next stage.

Filtering in

Print primes

```
bool is_prime(int n);

template <typename Execution>
void print_primes(const Execution & ex, int n)
{
  grppi::pipeline(exec,
    [i=0,max=n]() mutable -> optional<int> {
      if (i<=n) return i++;
      else return {};
    },
    grppi::keep(is_prime),
    [](int x) { cout << x << "\n"; }
  );
}
```

Filtering out

Discard words

```
template <typename Execution>
void print_primes(const Execution & ex, std::istream & is)
{
  grppi::pipeline(exec,
    [& file ]() -> optional<string> {
      string word;
      file >> word;
      if (! file ) { return {}; }
      else { return word; }
    },
    grppi::discard([](std::string w) { return w.length() < 4; }),
    [](std::string w) { cout << x << "\n"; }
  );
}
```

Stream reduction pattern

- A **stream reduction** pattern performs a reduction over the items of a subset of a data stream.

Stream reduction pattern

- A **stream reduction** pattern performs a reduction over the items of a subset of a data stream.

- **Key elements**
  - **window-size**: Number of elements in a reduction window.
  - **offset**: Distance between two consecutive window starts.
  - **identity**: Value used as identity in reductions.
  - **combiner**: Combination operation used for reductions.

Windowed reductions

Chunked sum

```
template <typename Execution>
void print_primes(const Execution & ex, int n)
{
  grppi::pipeline(exec,
    [i=0,max=n]() mutable -> optional<double> {
      if (i<=n) return i++;
      else return {};
    },
    grppi::reduce(100, 50, 0.0,
      [](double x, double y) { return x+y; }),
    [](int x) { cout << x << "\n"; }
  );
}
```

Stream iteration pattern

- A **stream iteration** pattern allows loops in data stream processing.
  - An operation is applied to a data item until a predicate is satisfied.
  - When the predicate is met, the result is sent to the output stream.

Stream iteration pattern

- A **stream iteration** pattern allows loops in data stream processing.
  - An operation is applied to a data item until a predicate is satisfied.
  - When the predicate is met, the result is sent to the output stream.

- **Key elements**:
  - A **transformer** that is applied to a data item on each iteration.
  - A **predicate** to determine when the iteration has finished.

Iterating

Print values $2^n * x$

```
template <typename Execution>
void print_values(const Execution & ex, int n)
{
  auto generator = [i=1,max=n+1]() mutable -> optional<int> {
    if (i<max) return i++;
    else return {};
  };

  grppi::pipeline(ex,
    generator,
    grppi::repeat_until(
      [](int x) { return 2*x; },
      [](int x) { return x>1024; }
    ),
    [](int x) { cout << x << endl; }
  );
}
```

Contexts

- A context allows to use a different execution policy for a subset of a pipeline.
  - Simplifies composition of pipelines.
  - Allocate sets of threads to different sub pipelines.

Execution contexts

```
template <typename E1, typename E2, typename Generator>
void run_pipe(const E1 & ex1, const E2 & ex2, Generator gen)
{
  grppi::pipeline(ex1,
    gen,
    run_with(ex2, pipeline(
      [](int x) -> double { return x*x; },
      [](double x) { return 1/x; },
    )),
    [](double x) { cout << x << "\n"; }
  );
}
```

Example

- Video frame processing for border detection with two filters.
  - Gaussian blur.
  - Sobel.

Example

- Video frame processing for border detection with two filters.
  - Gaussian blur.
  - Sobel.
- Using **pipeline** pattern:
  - S1: Frame reading.
  - S2: Gaussian blur (may apply **farm**).
  - S3: Sobel filter (may apply **farm**).
  - S4: Frame writing.

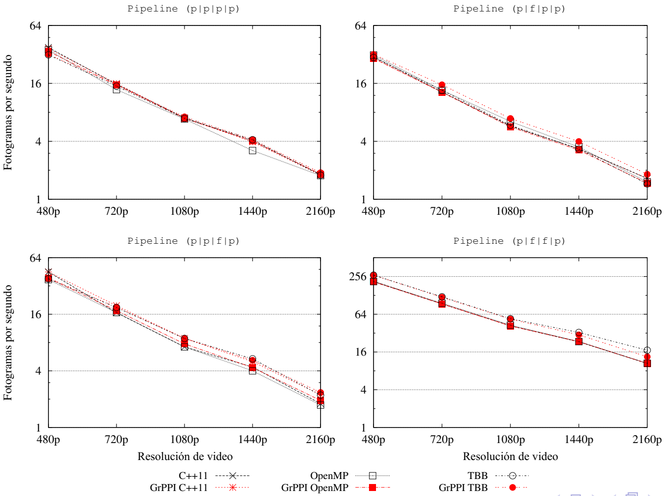
Example

- Video frame processing for border detection with two filters.
  - Gaussian blur.
  - Sobel.
- Using **pipeline** pattern:
  - S1: Frame reading.
  - S2: Gaussian blur (may apply **farm**).
  - S3: Sobel filter (may apply **farm**).
  - S4: Frame writing.
- **Approaches**:
  - Manual.
  - **GrPPI**.

Parallelization effort

Pipeline Composition	% LOC increase				
	C++ Threads	**OpenMP**	**Intel TBB**	**Fastflow**	**GrPPI**
(p | p | p | p)	+8.8 %	+13.0 %	+25.9 %	+16.5 %	+1.8 %
(p | f | p | p)	+59.4 %	+62.6 %	+25.9 %	+24.7 %	+3.1 %
(p | p | f | p)	+60.0 %	+63.9 %	+25.9 %	+24.7 %	+3.1 %
(p | f | f | p)	+106.9 %	+109.4 %	+25.9 %	+39.2 %	+4.4 %

Performance: frames per second

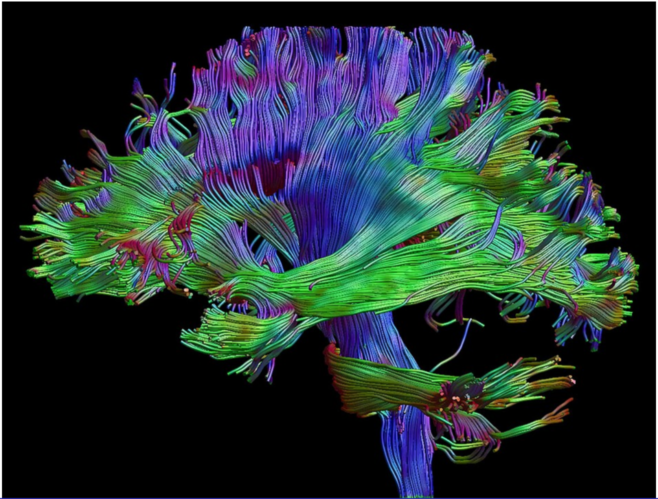


Brain MRI (Magnetich Resonance Imaging)

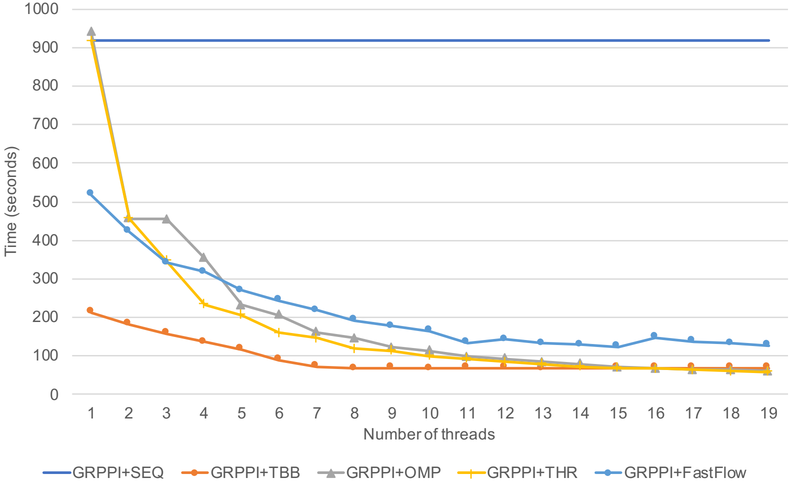
- Non intrusive method for getting internal anatomy.
- Huge amount of data generated.
- Applied to neuro-sciences.
  - Bipolar disorder.
  - Paranoia.
  - Schizophrenia.

- Identification of fibers and connectivity between areas in brain.

Fibbers in brain



MRI Evaluation



Summary

- An unified programming model for sequential and parallel modes.
- Multiple back-ends available.
- Current pattern set:
  - **Data**: **map**, **reduce**, **map/reduce**, **stencil**.
  - **Task**: **divide/conquer**.
  - **Streaming**: **pipeline** with nesting of **farm**, **filter**, **reduction**, **iteration**.

Future work

- Integrate additional backends (e.g. CUDA, OpenCL, MPI, ...).
- Eliminate metaprogramming by using Concepts.
- Extend and simplify the interface for data patterns.
- Better support of NUMA for native back-end.
- More patterns.
- More applications.

Recent publications

- **A Generic Parallel Pattern Interface for Stream and Data Processing**. D. del Rio, M. F. Dolz, J. Fernández, J. D. García. Concurrency and Computation: Practice and Experience. 2017.
- **Parallelizing and optimizing LHCb-Kalman for Intel Xeon Phi KNL processors**. P. Fernandez, D. Rio, M.F. Dolz, J. Fernández, O. Awile, J.D. Garcia, PDP 2018.
- **Supporting Advanced Patterns in GrPPI: a Generic Parallel Pattern Interface**. D. R. del Astorga, M. F. Dolz, J. Fernandez, and J. D. Garcia, Auto-DaSP 2017 (Euro-Par 2017).
- **Probabilistic-Based Selection of Alternate Implementations for Heterogeneous Platforms**. J. Fernandez, A. Sanchez, D. del Río, M. F. Dolz, J. D Garcia. ICA3PP 2017. 2017.
- **A C++ Generic Parallel Pattern Interface for Stream Processing**. D. del Río, M. F. Dolz, L. M. Sanchez, J. Garcia-Blas and J. D. Garcia. ICA3PP 2016.
- **Finding parallel patterns through static analysis in C++ applications**. D. R. del Astorga, M. F. Dolz, L. M. Sanchez, J. D. Garcia, M. Danelutto, and M. Torquati, International Journal of High Performance Computing Applications, 2017.

GrPPI

https://github.com/arcosuc3m/grppi

An introduction to **GrPPI**

Generic Reusable Parallel Patterns Interface

J. Daniel Garcia

ARCOS Group
University Carlos III of Madrid
Spain

February 2018