An introduction to **GrPPI**

Generic Reusable Parallel Patterns Interface

J. Daniel Garcia

ARCOS Group
University Carlos III of Madrid
Spain

February 2018
Warning

This work is under Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) license.

You are free to Share — copy and redistribute the material in any medium or format.

You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

You may not use the material for commercial purposes.

If you remix, transform, or build upon the material, you may not distribute the modified material.
An introduction to GrPPI

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 committee.
The GrPPI library has been partially supported by:


An introduction to GrPPI

GrPPI team

- **Main team**
  - J. Daniel Garcia (UC3M, lead).
  - David del Río (UC3M).
  - Manuel F. Dolz (UC3M).
  - Javier Fernández (UC3M).
  - Javier García Blas (UC3M).

- **Cooperation**
  - Plácido Fernández (UC3M-CERN).
  - Marco Danelutto (Univ. Pisa)
  - Massimo Torquati (Univ. Pisa)
  - Marco Aldinucci (Univ. Torino)
  - Fabio Tordini (Univ. Torino)
  - ...
1 Introduction

2 Simple use

3 Patterns in GrPPI

4 Evaluation

5 Conclusions
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).
An introduction to GrPPI

Introduction

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**.
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

```cpp
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

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

numvec add(const numvec & v1, const numvec & v2) {
    numvec res;
    res.reserve(v1.size()); // Assuming 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

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

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

Introduction

Adding two vectors

The STL way

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

numvec add(const numvec & v1, const numvec & v2) {
    numvec res;
    res.reserve(v1.size()); // Assume 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):
  - **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.
An introduction to GrPPI

Introduction

GrPPI ideals

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

---

cbed – J. Daniel Garcia – ARCOS@UC3M (josedaniel.garcia@uc3m.es) – Twitter: @jdgarciauc3m
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/grpippi
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 teaching tool

**SUMMARY and OUTLOOK**

- GrPPI enriched with FastFlow back-end
- No significant overhead added
- Currently not possible in GrPPI to deeply optimize particular compositions of nested patterns
- More tests using real-world applications are needed
- More patterns have to be ported (e.g. DSP patterns)
- GrPPI is currently used to teach parallel programming at University of PISA (SPM course -- Prof. M. Danelutto)
1 Introduction

2 Simple use

3 Patterns in GrPPI

4 Evaluation

5 Conclusions
Example: Transforming a sequence

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

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

![Diagram showing transformation of frames](image)
An introduction to GrPPI

Simple use

Transforming a sequence

Traditional explicit loop

```cpp
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**

```cpp
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)

```cpp
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)

```cpp
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;
}
```
An introduction to GrPPI

Simple use

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;
}
```
1. Introduction
2. Simple use
3. Patterns in GrPPI
4. Evaluation
5. Conclusions
3 Patterns in GrPPI
- Controlling execution
- Patterns overview
- Data patterns
- Task Patterns
- Streaming patterns
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**.
Some execution types allow finer configuration.

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

Interface:

```cpp
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**.
3 Patterns in GrPPI

- Controlling execution
- Patterns overview
- Data patterns
- Task Patterns
- Streaming patterns
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
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 data set, generating a new data set.
Maps on data sequences

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

- **Unidimensional:**

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

- **Unidimensional:**

- **Multidimensional:**
An introduction to GrPPI

Patterns in GrPPI

Data patterns

Single sequences mapping

Double all elements in vector sequentially

```cpp
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;
}
```
Double all elements in vector with OpenMP

```cpp
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;
}
```
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;
}
Add three vectors

```cpp
template <typename Execution>
std::vector<double> add_vectors(const Execution & ex,
конст std::vector<double> & v1,
конст std::vector<double> & v2,
конст 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

```cpp
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;
}
```
A **reduce** pattern combines all values in a data set using a binary combination operation.
Homogeneous reductions

Add a sequence of values

```cpp
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; });
}
```
An introduction to GrPPI

Patterns in GrPPI

Data patterns

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
An introduction to GrPPI
Patterns in GrPPI
Data patterns

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; }
    );
}
An introduction to GrPPI

Patterns in GrPPI

Data patterns

Heterogeneous reductions with map/reduce

Add areas of shapes

```cpp
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

```cpp
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)
{
  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;
                          });
}
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.

```cpp
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;
                           });
```
An introduction to GrPPI

Patterns in GrPPI

Data patterns

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

```cpp
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) { return {w, 1}; },
                               [](dictionary & lhs, const dictionary & rhs) { for (auto & entry : rhs) { lhs[entry.first] += entry.second; } return lhs; });
}
```
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.

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

```cpp
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;
        });
}
```
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(v)) > 1) r.push_back(*next(it));
                       return r;
                   })
    return res;
}
3 Patterns in GrPPI

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

```cpp
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

```cpp
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};
        }) ;
}
```
A patterned merge/sort

Ranges on vectors

```cpp
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

```cpp
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};
        });
}
```
3 Patterns in GrPPI

- Controlling execution
- Patterns overview
- Data patterns
- Task Patterns
- Streaming patterns
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.
A standalone pipeline is a top-level pipeline. Invoking the pipeline translates into its execution.
A **standalone pipeline** is a top-level pipeline. Invoking the pipeline translates into its execution.

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

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

```cpp
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 \( \text{bool} \)
    \[
    \text{if} \ (x) \ \{ \ /* \ ... \ */ \ \} \\
    \text{if} \ (!x) \ \{ \ /* \ ... \ */ \ \} 
    \]
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 dereferenced
    ```
    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 \( \text{bool} \)
    
    \[
    \begin{align*}
    &\text{if } (x) \{ /* ... */ \} \\
    &\text{if } (!x) \{ /* ... */ \}
    \end{align*}
    \]
  - Can be dereferenced
    
    \[ \text{auto } \text{val} = *x; \]
- The standard library offers an excellent candidate
  
  \[ \text{std::experimental::optional}<T> \]. \]
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

Simple pipeline

\[ x \rightarrow x^2 \rightarrow 1/x \rightarrow \text{print} \]

```cpp
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"; }
  ) ;
}
```

---

J. Daniel Garcia – ARCOS@UC3M (josedaniel.garcia@uc3m.es) – Twitter: @jdgarciauc3m
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

```cpp
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); });
}
```
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

Nested pipelines

Image processing

```cpp
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(in_file);
            if (!f) 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); }) ;
}
```
Image processing

```cpp
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;
        },
        [] (const frame & f) { return filter(f); },
        [] (const frame & f) { return gray_scale(f); },
        [& out_file] (const frame & f) { write_frame(out_file, f); }
    );
}
```
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

Nested pipelines

Image processing

```cpp
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); }
    );
}
```
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

Nested pipelines

Image processing

```c++
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); }
    ) ;
}
```

An introduction to **GrPPI**

Patterns in GrPPI

Streaming patterns

### Piecewise pipelines

#### Image processing

```cpp
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);
}
```
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

Piecewise pipelines

Image processing

```cpp
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) ;
}
```
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);
}

Image processing
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

### Piecewise pipelines

#### Image processing

```cpp
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);
}
```

---

[cited from: J. Daniel Garcia – ARCOS@UC3M (josedaniel.garcia@uc3m.es) – Twitter: @jdgarciauc3m]
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

Piecewise pipelines

Image processing

```cpp
void process(std::istream & in_file, std::ostream & out_file) {
    auto reader = [& in_file](){
        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.
Improving a video

```cpp
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) return {};
            return f;
        },
        farm(4, [] (const frame & f) { return improve(f); }),
        [& fileout] (const frame & f) { write_frame(f); }
    );
}
```
Improving a video

declare <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) return {};
        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);
}
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

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.
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

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"; }
    );
}
Discard words

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

An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

Filtering out
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

```cpp
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.
An introduction to GrPPI

Patterns in GrPPI

Streaming patterns

Iterating

Print values $2^n \times x$

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; }
    );
}
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

```cpp
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"; }   
    )  
}
```
1. Introduction
2. Simple use
3. Patterns in GrPPI
4. Evaluation
5. Conclusions
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.
An introduction to GrPPI

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**:
  - GrPPI.
Parallelization effort

<table>
<thead>
<tr>
<th>Pipeline Composition</th>
<th>% LOC increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C++ Threads</td>
</tr>
<tr>
<td>(p</td>
<td>p</td>
</tr>
<tr>
<td>(p</td>
<td>f</td>
</tr>
<tr>
<td>(p</td>
<td>p</td>
</tr>
<tr>
<td>(p</td>
<td>f</td>
</tr>
</tbody>
</table>
An introduction to GrPPI

Evaluation

Performance: frames per second

Pipeline (p|p|p|p)

Pipeline (p|f|p|p)

Pipeline (p|p|f|p)

Pipeline (p|f|f|p)

C++11
GrPPI C++11
OpenMP
GrPPI OpenMP
TBB
GrPPI TBB

Resolución de video

Fotogramas por segundo

Resolución de video

Fotogramas por segundo
An introduction to GrPPI Evaluation

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

![Graph showing time (seconds) vs. number of threads for different GrPPI techniques. The graph compares GrPPI+SEQ, GrPPI+TBB, GrPPI+OMP, GrPPI+THR, and GrPPI+FastFlow. The y-axis represents time in seconds, and the x-axis represents the number of threads. The graph shows improvement in time as the number of threads increases for all techniques.]
1 Introduction

2 Simple use

3 Patterns in GrPPI

4 Evaluation

5 Conclusions
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


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