GrPPI

Generic Reusable Parallel Patterns Interface

ARCOS Group
University Carlos III of Madrid
Spain

January 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**.
- **Improving applications**:
  - **REPARA**: Reengineering and Enabling Performance and poweR of Applications. Financiado por Comisión Europea (FP7). 2013–2016
  - **RePhrase**: REfactoring Parallel Heterogeneous Resource Aware Applications. Financiado por Comisión Europea (H2020). 2015–2018
- **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)
  - ...

1 Introduction

2 Data patterns

3 Task Patterns

4 Streaming patterns

5 Writing your own execution

6 Evaluation

7 Conclusions

1 Introduction
- Parallel Programming
- Design patterns and parallel patterns
- GrPPI architecture

Thinking in Parallel is hard.

Thinking is hard.

Yale Patt

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.
- **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.
- **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?

1 Introduction
- Parallel Programming
- Design patterns and parallel patterns
- GrPPI architecture

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

A brief history of patterns

- From building and architecture (Cristopher Alexander):
  - **1977**: A Pattern Language: Towns, Buildings, Construction.
  - **1979**: The timeless way of buildings.

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.

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.

1 Introduction
- Parallel Programming
- Design patterns and parallel patterns
- GrPPI architecture

Some ideals

- Applications should be expressed independently of the execution model.

Some ideals

- Applications should be expressed independently of the execution model.
- Multiple back-ends should be offered with simple switching mechanisms.

Some ideals

- Applications should be expressed independently of the execution model.
- Multiple back-ends should be offered with simple switching mechanisms.
- Interface should integrate seamlessly with modern C++ standard library.

Some ideals

- Applications should be expressed independently of the execution model.
- Multiple back-ends should be offered with simple switching mechanisms.
- Interface should integrate seamlessly with modern C++ standard library.
- Make use of modern (C++14) 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.

Setting up GrPPI

- **Structure.**
  - **include**: Include files.
  - **unit_tests**: Unit tests using GoogleTest.
  - **samples**: Sample programs.
  - **cmake-modules**: Extra CMake scripts.

Setting up GrPPI

- **Structure.**
  - **include**: Include files.
  - **unit_tests**: Unit tests using GoogleTest.
  - **samples**: Sample programs.
  - **cmake-modules**: Extra CMake scripts.

- Initial setup

```
mkdir build
cd build
cmake ..
make
```

CMake variables

- **GRPPI_UNIT_TESTS_ENABLE**: Enable building unit tests.
- **GRPPI_OMP_ENABLE**: Enable OpenMP back-end.
- **GRPPI_TBB_ENABLE**: Enable Intel TBB back-end.
- **GRPPI_EXAMPLE_APPLICATIONS_ENABLE**: Enable building example applications.
- **GRPPI_DOXY_ENABLE**: Enable documentation generation.

Execution policies

- The execution model is encapsulated by execution values.
- Current execution types:
  - **sequential_execution**.
  - **parallel_execution_native**.
  - **parallel_execution_omp**.
  - **parallel_execution_tbb**.
  - **dynamic_execution**.
- All top-level patterns take one *execution* object.

Concurrency degree

- Sets the number of underlying threads used by the execution implementation.
  - **sequential_execution** ⇒ 1
  - **parallel_execution_native** ⇒ **hardware_concurrency()**.
  - **parallel_execution_omp** ⇒ **omp_get_num_threads()**.

- **API**
  - **ex.set_concurrency_degree(4)**
  - **int n = ex.concurrency_degree()**

Dynamic back-end

- Useful if you want to take the decision at run-time.
- Holds any other execution policy (or empty).

Dynamic back-end

- Useful if you want to take the decision at run-time.
- Holds any other execution policy (or empty).

Selecting the execution back-end

```
grppi::dynamic_execution execution_mode(const std::string & opt) {
  using namespace grppi;
  if ("seq" == opt) return sequential_execution{};
  if ("thr" == opt) return parallel_execution_native{};
  if ("omp" == opt) return parallel_execution_omp{};
  if ("tbb" == opt) return parallel_execution_tbb{};
  return {};
}
```

Function objects

- GrPPI is heavily based on passing code sections as function objects (aka *functors*).

- **Alternatives**:
  - Standard C++ predefined functors (e.g. **std::plus<int>**).
  - Custom hand-written function objects.
  - Lambda expressions.

- Usually lambda expressions lead to more concise code.

1 Introduction

2 Data patterns

3 Task Patterns

4 Streaming patterns

5 Writing your own execution

6 Evaluation

7 Conclusions

2 Data patterns

- Map pattern
- Reduce pattern
- Map/reduce pattern
- Stencil pattern

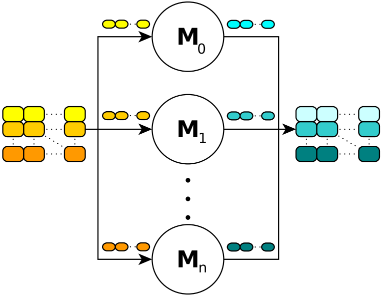
Maps on data sequences

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

- Given:
  - A sequence $x_1^1, x_2^1, \ldots, x_N^1 \in T_1$,
  - A sequence $x_1^2, x_2^2, \ldots, x_N^2 \in T_2$,
  - $\ldots$, and
  - A sequence $x_1^M, x_2^M, \ldots, x_N^M \in T_M$,
  - A function $f : T_1 \times T_2 \times \ldots \times T_M \mapsto U$

- It generates the sequence
  - $f(x_1^1, x_1^2, \ldots, x_1^M), f(x_2^1, x_2^2, \ldots, x_2^M), \ldots, f(x_N^1, x_N^2, \ldots, x_N^M)$

Maps on data sequences



Unidimensional maps

- **map** pattern on a single input data set.
- Given:
  - A sequence $x_1, x_2, \ldots, x_N \in T$
  - A function $f : T \mapsto U$
- It generates the sequence:
  - $f(x_1), f(x_2), \ldots, f(x_N)$

Key element

- **Transformer operation**: Any operation that can perform the transformation for a data item.

Key element

- **Transformer operation**: Any operation that can perform the transformation for a data item.
- **UnaryTransformer**: Any C++ callable entity that takes a data item and returns the transformed value.

```
auto square = [](auto x) { return x*x; };
auto length = [](const std::string & s) { return s.lenght(); };
```

Key element

- **Transformer operation**: Any operation that can perform the transformation for a data item.

- **UnaryTransformer**: Any C++ callable entity that takes a data item and returns the transformed value.

```
auto square = [](auto x) { return x*x; };
auto length = [](const std::string & s) { return s.lenght(); };
```

- **MultiTransformer**: Any C++ callable entity that takes multiple data items and return the transformed vaue.

```
auto normalize = [](double x, double y) { return sqrt(x*x+y*y); };
auto min = [](int x, int y, int z) { return std::min(x,y,z); }
```

Single sequences mapping

Double all elements in sequence

```
template <typename Execution>
std::vector<double> double_elements(const Execution & ex,
                                    const std::vector<double> & v)
{
  std::vector<double> res(v.size());
  grppi::map(ex, v.begin(), v.end(), res.begin(),
    [](double x) { return 2*x; });
}
```

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, v1.begin(), v1.end(), res.begin(),
    [](double x, double y) { return x+y; },
    v2.begin());
}
```

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, v1.begin(), v1.end(), res.begin(),
    [](double x, double y, double z) { return x+y+z; },
   v2.begin(), v3.begin());
}
```

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, re.begin(), re.end(), res.begin(),
    [](double r, double i) -> complex<double> { return {r,i}; }
    im.begin());
}
```

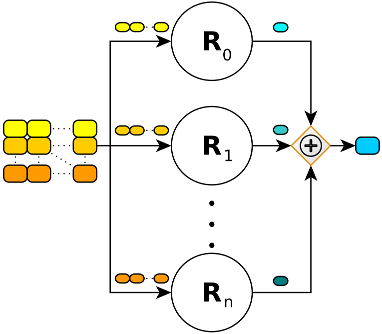
2 Data patterns

- Map pattern
- Reduce pattern
- Map/reduce pattern
- Stencil pattern

Reductions on data sequences

- A **reduce** pattern combines all values in a data set using a binary combination operation.
- Given:
  - A sequence $x_1, x_2, \ldots, x_N \in T$.
  - An identity value $id \in I$.
  - A combine operation $c : I \times T \mapsto I$
    - $c(c(x,y),z) \equiv c(x,c(y,z))$
    - $c(id,x) = \bar{x}$, where $\bar{x}$ is the value of x in I.
    - $c(id,c(id,x)) = c(id,x)$
    - $c(c(c(id,x),y),c(c(id,z),t)) = c(c(c(c(id,x),y),z),t)$
- It generates the value:
  - $c(\ldots c(c(id,x_1),x_2)\ldots,x_N)$

Reductions on data sequences



Homogeneous reduction

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; }) ;
}
```

Heterogeneous reduction

Add lengths of sequence of strings

```
template <typename Execution>
int add_lengths(const Execution & ex, const std::vector<std::string> & words)
{
  return grppi::reduce(words.begin(), words.end(), 0,
    []( int n, std::string w) { return n + w.length(); });
}
```

2 Data patterns

- Map pattern
- Reduce pattern
- Map/reduce pattern
- Stencil pattern

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 with single data set

- A **map/reduce** on a single input sequence producing a value.

Map/reduce with single data set

- A **map/reduce** on a single input sequence producing a value.
- Given:
  - A sequence $x_1, x_2, \ldots x_N \in T$
  - A mapping function $m : T \mapsto R$
  - A reduction identity value $id \in I$.
  - A combine operation $c : I \times R \mapsto I$

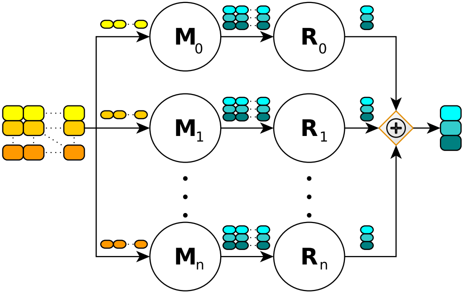
Map/reduce with single data set

- A **map/reduce** on a single input sequence producing a value.

- Given:
  - A sequence $x_1, x_2, \ldots x_N \in T$
  - A mapping function $m : T \mapsto R$
  - A reduction identity value $id \in I$.
  - A combine operation $c : I \times R \mapsto I$

- It generates a value reducing the mapping:
  - $c(c(c(id, m_1), m_2), \ldots, m_M)$
  - Where $m_k = m(x_k)$

Map/reduce pattern



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; }
  );
}
```

Map/reduce in multiple data sets

- A **map/reduce** on multiple input sequences producing a single value.

Map/reduce in multiple data sets

- A **map/reduce** on multiple input sequences producing a single value.
- Given:
  - A sequence $x_1^1, x_2^1, \ldots x_N^1 \in T_1$
  - A sequence $x_1^2, x_2^2, \ldots x_N^2 \in T_2$
  - $\ldots$
  - A sequence $x_1^M, x_2^M, \ldots x_N^M \in T_M$
  - A mapping function $m : T_1 \times T_2 \times \ldots \times T_M \mapsto R$
  - A reduction identity value $id \in I$.
  - A combine operation $c : I \times R \mapsto I$

Map/reduce in multiple data sets

- A **map/reduce** on multiple input sequences producing a single value.
- Given:
  - A sequence $x_1^1, x_2^1, \dots x_N^1 \in T_1$
  - A sequence $x_1^2, x_2^2, \dots x_N^2 \in T_2$
  - $\dots$
  - A sequence $x_1^M, x_2^M, \dots x_N^M \in T_M$
  - A mapping function $m : T_1 \times T_2 \times \dots \times T_M \mapsto R$
  - A reduction identity value $id \in I$.
  - A combine operation $c : I \times R \mapsto I$
- It generates a value reducing the mapping:
  - $c(c(c(id, m_1), m_2), \dots, m_M)$
  - Where $m_k = m(x_1^k, x_2^k, \dots, x_N^k)$

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, begin(v1), end(v1), 0.0,
    [](double x, double y) { return x*y; },
    [](double x, double y) { return x+y; },
   v2.begin());
}
```

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;
    });
}
```

2 Data patterns

- Map pattern
- Reduce pattern
- Map/reduce pattern
- Stencil pattern

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*.

Stencil with single data set

- A **stencil** on a single input sequence producing an output sequence.

Stencil with single data set

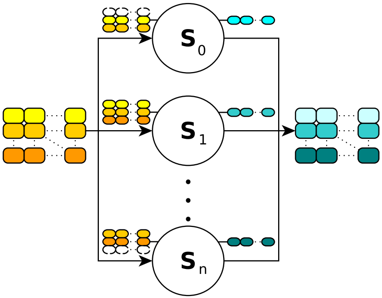
- A **stencil** on a single input sequence producing an output sequence.

- Given:
  - A sequence $x_1, x_2, \ldots, x_N \in T$
  - A neighbourhood function $n : I \mapsto N$
  - A transformation function $f : I \times N \mapsto U$

Stencil with single data set

- A **stencil** on a single input sequence producing an output sequence.
- Given:
  - A sequence $x_1, x_2, \ldots, x_N \in T$
  - A neighbourhood function $n : I \mapsto N$
  - A transformation function $f : I \times N \mapsto U$
- It generates the sequence:
  - $f(n(x_1)), f(n(x_2)), \ldots, f(n(x_N))$

Stencil pattern



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;
}
```

Stencil with multiple data sets

- A **stencil** on multiple input sequences producing an output sequence.

Stencil with multiple data sets

- A **stencil** on multiple input sequences producing an output sequence.
- Given:
  - A sequence $x_1^1, x_2^1, \ldots, x_N^1 \in T_1$
  - A sequence $x_1^2, x_2^2, \ldots, x_N^2 \in T_1$
  - ...
  - A sequence $x_1^M, x_2^M, \ldots, x_N^M \in T_1$
  - A neighbourhood function $n : I_1 \times I_2 \times I_M \mapsto N$
  - A transformation function $f : I_1 \times N \mapsto U$

Stencil with multiple data sets

- A **stencil** on multiple input sequences producing an output sequence.
- Given:
  - A sequence $x_1^1, x_2^1, \ldots, x_N^1 \in T_1$
  - A sequence $x_1^2, x_2^2, \ldots, x_N^2 \in T_1$
  - $\ldots$
  - A sequence $x_1^M, x_2^M, \ldots, x_N^M \in T_1$
  - A neighbourhood function $n : I_1 \times I_2 \times I_M \mapsto N$
  - A transformation function $f : I_1 \times N \mapsto U$
- It generates the sequence:
  - $f(n(x_1)), f(n(x_2)), \ldots, f(n(x_N))$

Multiple sequences stencil

Neighbour average

```
template <typename It>
std::vector<double> get_around(It i, It first, It last) {
  std::vector<double> r;
  if (i!= first) r.push_back(*std::prev(i));
  if (std::distance(i, last)>1) r.push_back(*std::next(i));
}

template <typename Execution>
std::vector<double> neib_avg(const Execution & ex, const std::vector<double> & v1,
                             const std::vector<double> & v2)
{
  std::vector<double> res(std::min(v1.size(),v2.size()));
  grppi::stencil(ex, v.begin(), v.end(),
    [](auto it, auto n) { return *it + accumulate(begin(n), end(n)); },
    [&](auto it, auto it2) {
      vector<double> r = get_around(it1, v1.begin(), v1.end());
      vector<double> r2 = get_around(it2, v2.begin(), v2.end());
      copy(r2.begin(), r2.end(), back_inserter(r));
      return r;
    },
    v2.begin());
  return res;
}
```

1 Introduction

2 Data patterns

3 Task Patterns

4 Streaming patterns

5 Writing your own execution

6 Evaluation

7 Conclusions

3 Task Patterns

- Divide/conquer pattern

Divide/conquer pattern

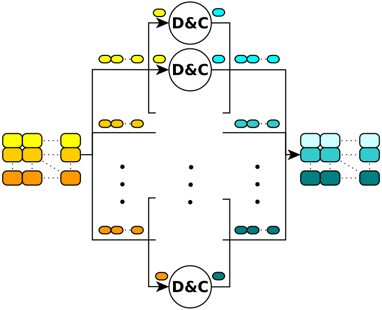
- A **divide/conquer** pattern splits a problem into two or more independent subproblems until a base case is reached.
  - The base case is solved directly.
  - The results of the subproblems are combined until the final solution of the original problem is obtained.

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.

Divide/conquer pattern



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> 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),
    [](auto r) -> vector<range> {
      if (1>=r.size()) return {r};
      else return divide(r);
    },
    [](auto x) { return x; },
    [](auto r1, auto r2) {
      std::inplace_merge(r1.first, r1.last, r2.last);
      return range{r1.first, r2.last};
    });
}
```

1 Introduction

2 Data patterns

3 Task Patterns

4 Streaming patterns

5 Writing your own execution

6 Evaluation

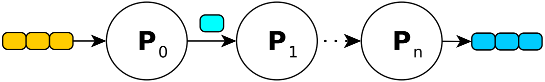
7 Conclusions

4 Streaming patterns

- Pipeline pattern
- Execution policies and pipelines
- Farm stages
- Filtering stages
- Reductions in pipelines
- Iterations in pipelines

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

Standalone pipeline

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

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

- For every non-empty value generated by g, it evaluates:
  - $f_n(f_{n-1}(\ldots f_1(g())))$

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

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 {};
    },
    grppi::pipeline(
      [](int x) -> double { return x*x; },
      [](double x) { return 1/x; }),
    [](double x) { cout << x << "\n"; }
  );
}
```

Piecewise pipelines

- A pipeline can be piecewise created.

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

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

  grppi::pipeline(ex, generator, inner, printer);
}
```

4 Streaming patterns

- Pipeline pattern
- Execution policies and pipelines
- Farm stages
- Filtering stages
- Reductions in pipelines
- Iterations in pipelines

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

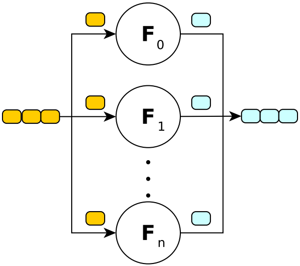
- Some policies (**native** and **omp**) use queues to communicate pipeline stages.
- **Properties**:
  - **Queue size**: Buffer size of the queue.
  - **Mode**: *blocking* versus *lock-free*.
- **API**
  - **ex.set_queue_attributes(100, mode::blocking)**

4 Streaming patterns

- Pipeline pattern
- Execution policies and pipelines
- **Farm stages**
- Filtering stages
- Reductions in pipelines
- Iterations in pipelines

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



Farms in pipelines

Square values

```
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  {};
    },
    grppi :: farm(4
      []( int  x)  −> double { return x∗x; }),
    []( double x) { cout << x << "\n";  }
  ) ;
}
```

Piecewise farms

Square values

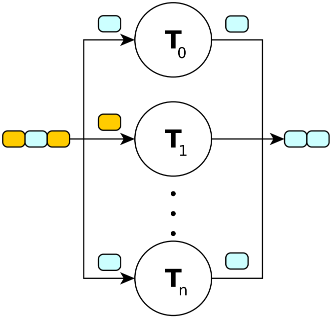
```
template <typename Execution>
void run_pipe(const Execution & ex, int n)
{
  auto inner = grppi::farm(4 [](int x) -> double { return x*x; });

  grppi::pipeline(ex,
    [i=0,max=n] () mutable -> optional<int> {
      if (i<max) return i;
      else return {};
    },
    inner,
    [](double x) { cout << x << "\n"; }
  );
}
```

4 Streaming patterns
- Pipeline pattern
- Execution policies and pipelines
- Farm stages
- Filtering stages
- Reductions in pipelines
- Iterations in pipelines

Filter pattern

- A **filter** pattern discards (or keeps) the data items from a data stream based on the outcome of a predicate.



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

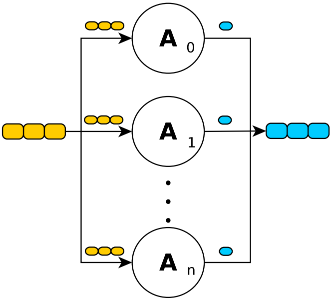
```
template <typename Execution>
void print_primes(const Execution & ex, std::istream & is)
{
  grppi :: pipeline (exec,
    [& file ]()  −> optional<string> {
      string word;
      file >> word;
      if (! file ) { return {}; }
      else { return word; }
    },
    grppi :: discard ([]( std ::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

4 Streaming patterns

- Pipeline pattern
- Execution policies and pipelines
- Farm stages
- Filtering stages
- Reductions in pipelines
- Iterations in pipelines

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 window reduction
  - **offset**: Distance between the begin of two consecutive windows
  - **identity**: Initial value used for reductions
  - **combiner**: 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";  }
  ) ;
}
```

4 Streaming patterns
- Pipeline pattern
- Execution policies and pipelines
- Farm stages
- Filtering stages
- Reductions in pipelines
- Iterations in pipelines

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; }
  );
}
```

1 Introduction

2 Data patterns

3 Task Patterns

4 Streaming patterns

5 Writing your own execution

6 Evaluation

7 Conclusions

Addine a new policy

- Adding a new execution policy is done by writing a new class.
  - No inheritance needed.
    - *"Inheritance is the base class of all evils"* (Sean Parent).
  - No dependency from the library.
  - Additionally configure some meta-functions (until we have concepts).

My custom execution

my_execution

```
class my_execution {
  my_execution() noexcept;

  void set_concurrency_degree(int n) const noexcept;
  void concurrency_degree() const noexcept;

  void enable_ordering() noexcept;
  void disable_ordering() noexcept;
  bool is_ordered() const noexcept;

  // ...
};

template <>
constexpr bool is_supported<my_execution>() { return true; }
```

Adding a pattern

my_execution::map

```
class my_execution {

  // ...

  template <typename ... InputIterators, typename OutputIterator,
            typename Transformer>
  constexpr void map(std::tuple<InputIterators...> firsts ,
      OutputIterator first_out , std :: size_t sequence_size,
      Transformer && transform_op) const;

  // ...
};

template <>
constexpr bool supports_map<my_execution>() { return true; }
```

Some helpers in the library

Applying a function to a tuple of iterators

```
template <typename F, typename ... Iterators, template <typename ...> class T>
decltype(auto) apply_deref_increment(
    F && f,
    T<Iterators ...>  &  iterators )
```

- Takes a function **f** and a tuple of iterators (e.g. result of **make_tuple(it1, it2, it3)**.
- Returns **f(*it1++, *it2++, *it3++)**.
- Very convenient for implementing data patterns.
- More like this in **include/common/iterator.h**.

Implementing map

map

```
template <typename ... InputIterators, typename OutputIterator,
          typename Transformer>
void my_execution_native::map(std::tuple<InputIterators...> firsts ,
    OutputIterator first_out , std :: size_t sequence_size,
    Transformer transform_op) const
{
  using namespace std;
  auto process_chunk = [&transform_op](auto fins, std::size_t size, auto fout)
  {
    const auto l = next(get<0>(fins), size);
    while (get<0>(fins)!=l) {
      *fout++ = apply_deref_increment(
          std :: forward<Transformer>(transform_op), fins);
    }
  };
  // ...
```

Implementing map

map

```
// ...
const int chunk_size = sequence_size / concurrency_degree_;
{
  some_worker_pool workers;
  for (int i=0; i!=concurrency_degree_-1; ++i) {
    const auto delta = chunk_size * i;
    const auto chunk_firsts = iterators_next(firsts,delta);
    const auto chunk_first_out = next(first_out, delta);
    workers.launch(process_chunk, chunk_firsts, chunk_size, chunk_first_out);
  }

  const auto delta = chunk_size * (concurrency_degree_ - 1);
  const auto chunk_firsts = iterators_next(firsts,delta);
  const auto chunk_first_out = next(first_out, delta);
  process_chunk(chunk_firsts, sequence_size - delta, chunk_first_out);
} // Implicit pool synch
}
```

1 Introduction

2 Data patterns

3 Task Patterns

4 Streaming patterns

5 Writing your own execution

6 Evaluation

7 Conclusions

Evaluation

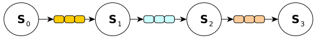
- **Plataform**:
  - 2 $\times$ Intel Xeon Ivy Bridge E5-2695 v2.
  - Total number of cores: 24.
  - Clock frequency: 2.40 GHz.
  - L3 cache size: 30 MB.
  - Main memory: 128 GB DDR3.
  - OS: Ubuntu Linux 14.04 LTS, kernel 3.13.
- **Software**:
  - Compiler: GCC 6.2.
  - OpenMP 4.0: included in GCC.
  - ISO C++ Threads: included in the C++ STL.
  - Intel TBB: `www.threadingbuildingblocks.org`

Use case

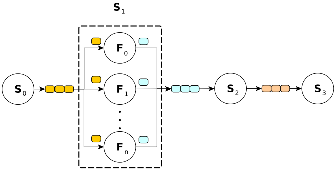
- Video processing application for detecting edges using the filters:
  - Gaussian Blur
  - Sobel operator
- It uses a **pipeline** pattern:
  - S1: Reading frames from a camera
  - S2: Apply the Gaussian Blur filter (it can use a **farm**)
  - S3: Apply the Sobel operator (it can use a **farm**)
  - S4: Writing frames into a file
- **Parallel variants**:
  - using the back ends directly
  - using

Pipeline compositions

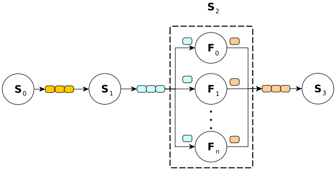
- Pipeline+farm compositions made in the video application:



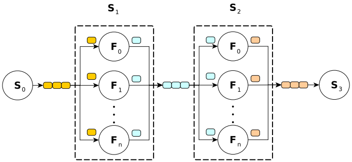
(a) Non-composed Pipeline.



(b) Pipeline (s | f | s | s).



(c) Pipeline (s | s | f | s).

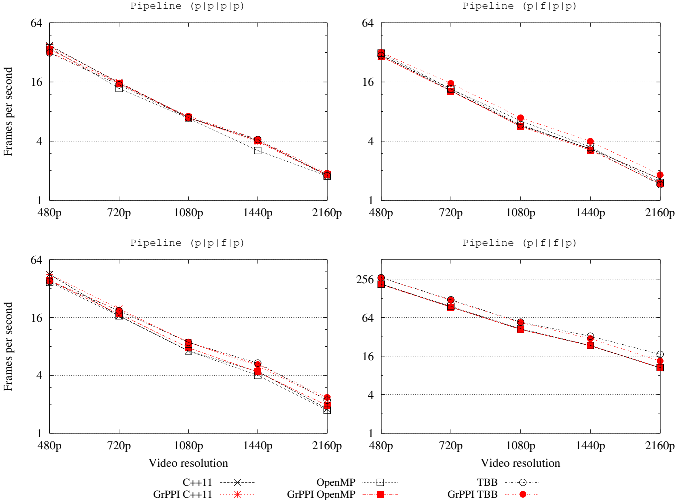


(d) Pipeline (s | f | f | s).

Usability of

Pipeline composition	% of increase of lines of code w.r.t sequential			
	C++ Threads	**OpenMP**	**Intel TBB**	
(p | p | p | p)	+8.8 %	+13.0 %	+25.9 %	+1.8 %
(p | f | p | p)	+59.4 %	+62.6 %	+25.9 %	+3.1 %
(p | p | f | p)	+60.0 %	+63.9 %	+25.9 %	+3.1 %
(p | f | f | p)	+106.9 %	+109.4 %	+25.9 %	+4.4 %

Performance: frames per second



Observations

- Using **farm** for both stages leads an improved FPS rate.
- Using **farm** for only one stage does not any bring significant improvement.
- Impact of on performance
  - Negligible overheads of about 2%
- Impact on programming efforts
  - Significant less efforts with respect to other programming models

1 Introduction

2 Data patterns

3 Task Patterns

4 Streaming patterns

5 Writing your own execution

6 Evaluation

7 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**.
- Current limitation:
  - Pipelines cannot be nested inside other patterns (e.g. iteration of a pipeline).

Future work

- Integrate additional backends (e.g. FastFlow, CUDA).
- Eliminate metaprogramming by using Concepts.
- Extend and simplify the interface for data patterns.
- Support multi-context 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.

- **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

GrPPI

Generic Reusable Parallel Patterns Interface

ARCOS Group
University Carlos III of Madrid
Spain

January 2018