Document Number: P0374R0
Date: 2016-05-30
Reply to: josedaniel.garcia@uc3m.es
Audience: SG1.

Stream parallelism patterns

J. Daniel Garcia
David del Rio
Manuel F. Dolz
Javier Garcia-Blas
Luis M. Sanchez

Computer Science and
Engineering Department
University Carlos III of Madrid

Marco Danelutto
Massimo Torquati

Computer Science Department
University of Pisa

1 Motivation

With the approval of the library parallelism extensions, first as a TS [1] and later as part of the upcoming C++17 standard [2], C++ programmers are able to use a number of generic parallel algorithms with low effort. However, all these algorithms help to solve what it is commonly known as *data parallel problems*.

Although the standard library offers a number of specific algorithms (e.g. `sort()`), it also offers more general algorithms that have been commonly referred as parallel patterns. Examples of those data parallel patters are `transform()`, `reduce()`, `inclusive_scan()`, and `exclusive_scan()`.

We think that those patterns should be complemented with other families of parallel patterns such as stream parallel patterns.

Stream parallel patterns have all in common that they process a stream of data and perform continuous processing on that data stream. In this initial paper, we focus on four specific patterns: *farm*, *pipeline*, *filter*, and *stream-reduce*.

2 Stream parallel patterns

Stream parallel patterns exploit parallelism in the processing of different items belonging to one or more input data streams. In general, an input data stream is characterized by having a elements of a given type and being able to provide items, one after the other, that will be used by some computation. While those patterns might be seen similar in some cases to existing traditional data parallel algorithms, a key difference is that neither the full sequence nor the number of items in the sequence are known in advance.

2.1 Approaches to manipulating streams

Two approaches can be considered to manipulate streams in different patterns:

- A data oriented approach: using data types for input and output streams.
- A more functional approach, through generation and consumption functions (or more generally, callable objects).

Using a data oriented approach can be achieved by using two types: a type for representing an input stream and a type for representing an output stream.

```
input_stream<int> s1 = get_input_stream();
output_stream<double> s2 = get_output_stream();

// Sequential processing of stream
```

```
while (!s1.finished()) {
  auto x = s1.get();
  auto y = compute(x);
  s2.put(y);
}
```

This approach could also be obtained by using types satisfying the concepts of `InputIterator` for input streams and `OutputIterator` for output streams. However, using iterators requires a mechanism to signal when the input stream has finished producing data.

A different approach could be representing the source and sink streams as callable objects. The source stream becomes a callable object returning an item each time. A protocol needs to be established to signal the end of the stream. One option would be to return a `pair` with a *value* and a `bool`. However, a different approach would be to return an `optional<value>`, signaling the end of stream with an empty object. Then, a source stream can be represented by a callable object that produce an `optional<value>`,

```
auto source = [&ifile] {
  if (! ifile ) return optional<int>{};
  else {
    int x;
    ifile >> x;
    return make_optional(x);
  }
};
```

and a sink stream can be represented by a callable object that takes an `optional<value>`.

```
auto sink = [&ofile] (optional<double> v) {
  if (v) ofile << *v:
}

// Sequential processing of stream
for (optional<int> && x = source(); x; x=source()) {
  auto y = compute(*x);
  sink(y);
}
```

2.2 Composing complex patterns

Being able to compose complex patterns from simpler ones is a highly desirable characteristic as it allows to express a complex stream computation by nesting simpler stream computation steps. Moreover, the ability to express a stream pattern in multiple ways provides the ability to perform transformations on the computation structures to improve performance [3].

To be able to compose patterns, we represent separately the idea of stream source and stream sink. Every top-level pattern may then take those streams to be able to get input data and store output data. We call those elements collectively the *bounds* of the computation.

```
bounds b{
  [& infile ] { return (! infile )?optional<int>{}:make_optional(read_value(infile)); }
  [& ofile ] (int x) { ofile << x << endl; }
};
```

Any pattern may take as an argument those bounds to interact with external streams.

```
patternA(par,
  bounds{
    [& infile ] { return (! infile )?optional<int>{}:make_optional(read_value(infile)); }
    [& ofile ] (int x) { ofile << x << endl; }
  },
  [] (int x) { return x/2; }
);
```

3 Farm

A farm, sometimes also referred as a *task farm* or a *stream map* (in reference to the functional and data parallel *map* pattern), performs a transformation on every item coming from an input stream and generates a new output stream of items. The computations performed during the transformation are considered fully independent one from the other.

There is a single key parameter to a *farm* pattern:

- A single transformation function operating on stream individual data.

Parallel execution of a *farm* pattern may be controlled by a number of optional parameters (with default values):

- **Parallelism degree**: Number of parallel threads of execution performing computations on different elements from the input stream and generating elements to the output stream.
- **Distribution policy**: Policy used to distribute input data items to computation execution threads. A possible strategy is performing *round-robin*. Another possible strategy is allowing each free execution thread to take data items (*auto-scheduling*).
- **Granularity**: Number of consecutive items taken by an execution thread. Appropriate granularity depends highly on the size of individual data items and the inter-arrival time.
- **Ordering**: Specifies if output needs to be ordered or can produce values in an unordered fashion (with regard to input arrival ordering).

Given an input stream with data items of type `T` and an output stream with data items of type `U`. A *farm transformer* is any callable object `f` where the statement:

```
T input;
U output = f(input);
```

is a valid statement.

Making use of a standalone *farm* pattern requires specifying a source, a transformer, and a sink.

```
int read_value(istream & is) {
  int x;
  is >> x;
  return x;
}

farm(par,
  bounds{
    [& infile ] { return (! infile )?optional<int>{}:make_optional(read_value(infile)); },
    [& outfile ] (double x) { ofile << x << endl; }
  },
  [] (int x) { return 1.0/x; }
);
```

A simpler form of *farm* can be specified for the cases where the source and sink are not specified. Note that this second form is useful only in compositions.

```
auto f1 = farm(par,
  [] (int x) { return 1.0/x; }
);
```

This form is useful in cases where the *farm* will be used inside another more complex pattern.

```
another_pattern(par,
  do_something,
  farm(par, [] (const vector<int> & v) { return max_element(begin(v), end(v)); },
  do_something_else
);
```

4 Pipeline

A *pipeline* performs a computation in several stages. The first stage takes data items from an input stream. Each stage takes data produced from previous stage and performs a transformation computation generating data items for the next stage. All those stages may be performed in parallel.

A *pipeline* takes a number of callable objects:

- The first callable object is a generator that does not take any argument and produces data items from first type T_1.
- Every intermediate stage take data items from type T_i and generates data items from type T_{i+1}.
- The last callable object is a consumer that takes data items from type T_n.

Given an input stream with data items of type T_0, an output stream of type T_n, and a number of intermediate transformation functions f_i, the following expressions are valid:

```
T0 x0;
T1 x1 = f1(x0);
T2 x2 = f2(x1);
//...
Tn xn = fn(xn-1);
```

Making use of a pipeline function requires specifying a number of intermediate transformation functions and source and sink callable objects.

```
pipeline(par,
  bounds{
    [& infile ] -> optional<frame> { return read_frame(infile); },
    [&outfile] (frame f) { write_frame( outfile , f); }
  },
  [] (frame f) { return filter1 (f); },
  [] (frame f) { return filter2 (f); }
);
```

A second form without source and sink allows composition.

```
farm(par,
  bounds{
    [& infile ] -> optional<frame> { return read_frame(infile); },
    [&outfile] (frame f) { write_frame( outfile , f); }
  },
  pipeline(par,
    [] (frame f) { return filter1 (f); },
    [] (frame f) { return filter2 (f); }
  )
);
```

5 Filter

The *filter* pattern selects data items from an input data stream according to a predicate so that only data items that satisfy it are sent to the output stream.

There is a single key parameter to the *filter* pattern:

- A predicate function operating on the stream individual data.

The parallel execution of a *filter* pattern may be controlled by the same parameters of the *farm* pattern.

Given an input stream with data items of type `T`, a *filter predicate* is any callable object `p` where the following statements are valid:

```
T x;
bool c = p(x);
```

Making use of a *filter* pattern requires specifying a generator, a consumer, and a predicate. In this case the data type returned by the generator function and the data type received by the consumer function must have compatible types.

```
filter (par,
  bounds{
    [& infile ] -> optional<frame> { return read_file(infile); },
    [& outfile ] (frame f) { write_frame( outfile , f ); }
  },
  [] (const frame & f) { return is_valid(f ); }
);
```

However, the most common use of the *filter* pattern is to act as an intermediate stage in a higher order pattern.

```
pipeline (par,
  bounds{
    [& infile ] -> optional<frame> { return read_file(infile); },
    [& outfile ] (frame f) { write_frame( outfile , f ); }
  },
  filter (par, [] (const frame & f) { return f.is_valid (); },
  farm(par, [] (const frame & f) { return f.enhance(); }
);
```

6 Stream reduce

The *stream-reduce* pattern applies a reduction operation on data items from input stream delivering result to an output stream.

Parameters for the *stream-reduce* pattern are:

- A *reduction* function which is a binary operation allowing to reduce two elements from the input stream.
- The *window size* which defines the number of input data items that are needed to produce an output data item.
- The *window distance* which defines the distance between the start of two consecutive windows.

Given an input stream with data items of type `T`, a *reduction* function is any callable object `r` where the following statements are valid:

```
T x, y, z;
z = r(x,y);
```

Making use of a *stream-reduce* pattern requires specifying a *reduction* function, a window size, and a window distance.

```
stream_reduce(par, 1000, 10,
  bounds{
    [& infile ] -> optional<int> { return (!infile)?optional<int>{}:make_optional(read_value(infile)); },
    [& outfile ] (int x) { write_value( outfile ,x); }
  },
  [] (int x, int y) { return max(x,y); }
);
```

As in previous case, *stream-reduce* is also suitable for composition:

```
farm(par,
  bounds{
    [& infile ] -> optional<int> { return (!infile)?optional<int>{}:make_optional(read_value(infile)); },
    [& outfile ] (int x) { write_value( outfile ,x); }
  },
  stream_reduce(par, 1000, 1000, [] (int x, int y) { return max(x,y); }
);
```

7 Implementation experience

There is a number of different library solutions providing stream parallel patterns. For example, the *FastFlow* [4] (`http://mc-fastflow.sourceforge.net/`) library offers all these patterns as a library with a more traditional API. Other examples include *StreamIt* (http://groups.csail.mit.edu/cag/streamit/) and Intel TBB (which offers partial support).

Acknowledgments

This work has received funding from the European Union Seventh Framework Programme (FP7/2007–2013) under grant agreement n. 609666 and from the European Union H2020 Programme under grant agreement n. 644235.

References

[1] ISO/IEC JTC1/SC22. Programming Languages – Technical Specification for C++ Extensions for Parallelism. International Technical Specification ISO/IEC 19570:2015, ISO, December 2015.

[2] ISO/IEC JTC1/SC22. The Parallelism TS Should be Standardized. Working Paper P0024R1, ISO/IEC JTC1/SC22/WG21, February 2016.

[3] Vladimir Janjic, Christopher Brown, Kevin Hammond, Kenneth Mackenzie, Marco Aldinucci, Marco Danelutto, and J. Daniel Garcia. RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Application. In *24th Euromicro International Conference on Parallel, Distributed and Network Based Processing (PDP 20126)*, Heraklion, Greece, February 2016.

[4] Marco Danelutto and Massimo Torquati. Structured parallel programming with "core" fastflow. In Viktória Zsók, Zoltán Horváth, and Lehel Csató, editors, *Central European Functional Programming School*, volume 8606 of *LNCS*, pages 29–75. Springer, 2015.