Document Number: N4167
Date: 2014-10-10
Authors: Grant Mercer
Agustín Bergé
Hartmut Kaiser

Transform Reduce, an Additional Algorithm for C++ Extensions for Parallelism.

Contents

1 General [algorithm.general]

1.1 Scope [algorithm.general.scope]

The goal of this paper is to widen the range of problems that the C++ standards proposal N4071, A Technical Specification for C++ Extensions for Parallelism, Revision 1 may encompass. Offering a greater use case for these algorithms may help further extend the significance of this Technical Specification.

This document describes and outlines the problems we have encountered with the current algorithm requirements, as specified in standards proposal N4071. The algorithm described in this document is a solution to a general set of problems that the current algorithms do not support. *Transform Reduce* is to be considered as an addition to the current set of algorithms in proposal N4071, not a replacement.

1.2 References [parallel.general.references]

The following reference is necessary for the contents of this document.

- ISO/IEC N4071:2014, Technical Specification for C++ Extensions for Parallelism

ISO/IEC N4071:2014 is the latest draft of a paper proposal to add parallelism to the C++ standard template library. The whole of the document is included into this document by reference.

2 Reduce Restrictions [algorithm.reduce.restrictions]

2.1 Parallel Reduce [algorithm.reduce.def]

According to the Technical Specification, the primary difference between **reduce** and **accumulate** is that the behavior of **reduce** may be non-deterministic for non-associative or non-commutative **operator+ .** The complexity of this algorithm is defined as *O*(last – first) applications of operator+ .

2.2 Use Cases Beyond Reduce [algorithm.reduce.use]

There are certain cases in which the Parallel Reduce as defined in the technical specification is not sufficient. One such problem is the calculation of a dot product: given a vector of tuples containing their respective X and Y values. Traditionally this could be accomplished using a simple *std::accumulate*:

Fig 1

```
struct Point {
    double x, y;
};

std::vector<Point> values(10007, Point{2.0, 2.0});

double result =
    std::accumulate(std::begin(values), std::end(values), 0.0,
        [](double result, Point curr)
        {
            return result + curr.x * curr.y;
        });
```

This is a straight forward and intuitive solution, but cannot be parallelized. Accumulate depends on the previously calculated result in order to progress, creating a data dependency that cannot be worked around. When attempting to solve this problem using the algorithms offered by the technical specification, *parallel::reduce* fits best. However due to the requirements of reduce a number of workarounds were created in order to solve the problem:

Fig 2

```
Point result =
    std::experimental::parallel::reduce(
        std::experimental::parallel::par,
        std::begin(values),
        std::end(values),
        Point{0.0, 0.0},
        [](Point res, Point curr)
        {
            return Point{
               res.x * res.y + curr.x * curr.y, 1.0};
        }
    );
```

The new dot product solution now resembles a reduce over matrices:

$\begin{bmatrix} a_x \\ a_y \end{bmatrix} * \begin{bmatrix} b_x \\ b_y \end{bmatrix} \Rightarrow \begin{bmatrix} C \\ 1 \end{bmatrix}$ where $\begin{bmatrix} a_x \\ a_y \end{bmatrix} * \begin{bmatrix} b_x \\ b_y \end{bmatrix} * \begin{bmatrix} c_x \\ c_y \end{bmatrix} = \begin{bmatrix} Z_{abc} \\ 1 \end{bmatrix}$ and Z being the result

There is another major issue with this solution however, an unneeded multiplication. The current requirements of Reduce are that the return type must be the same type as the iterator type value, forcing the result in this case to become a `Point`. The result is instead stored inside the x member; a double would be preferable but does not satisfy the requirements of reduce. Consequently the second element of the result must be multiplied at each iteration to prevent an invalid value when partitions join together. While the implementation works, this goes against the philosophy of how Reduce should be used and is only considered a workaround; this should not be a valid use case for Reduce.

2.3 Transform reduce [algorithm.reduce.transformreduce]

This document proposes to add an additional algorithm named *transform_reduce*. transform_reduce would solve all use cases in which the result type differs from the input type; such as the example in section **2.2 Fig 1**. The algorithm uses a *conversion* function specified by the user in order to convert the input type to a return type before applying a reduce. The requirements of the function follow as:

```
template <typename InIter, typename T, typename Reduce, typename Convert>
T transform_reduce(InIter first, InIter last, T init,
    Reduce red_op, Convert conv_op)
```

1. *Returns:* Sum of the given value and elements in the given range
2. *Requires:* The **operator+** function associated with `T` shall not invalidate iterators or subranges, nor modify elements in the range.
3. *Complexity:* O(first – last)

Previously In section **2.2 Fig 2**, the final result of the dot product example was required to be a `Point` although only needing one field; creating unneeded multiplication. The return type `T` of *transform_reduce* allows the dot product solution to now be written as:

Fig 3

```
double result =
    std::experimental::parallel::transform_reduce(
        std::experimental::parallel::par,
        std::begin(values),
        std::end(values),
        0.0,
        std::plus<double>(),
        [](Point r)
        {
            return r.x * r.y;
        }
    );
```

This solution is better for a number of reasons: the code does not present any workarounds, it conforms to the requirements of *Transform Reduce,* there are no unnecessary operations taking place in the execution and the code looks much cleaner than previous solutions. Any other problems that are similar to dot product or have special requirements can benefit greatly from *Transform Reduce*.