Document number: *N3429=12-0119.*
Date: *2012-09-21*
Reply-to: Artur Laksberg < arturl@microsoft.com>
Herb Sutter < hsutter@microsoft.com>
Arch D. Robison, Intel Corp., <arch.robison@intel.com>
Sana Mithani <sanam@microsoft.com>

A C++ Library Solution to Parallelism

Contents

II. Introduction

We are proposing a high-level cross-platform parallel programming model for C++. This proposal provides an overview of several parallel algorithms to be added to the C++ standard.

III. Motivation and Scope

With the end of Moore's law and increasing processor clock speeds, advances in C++ are necessary to maintain high performance and scalability in programs. Any speed up required in applications needs to make efficient use of multicore processors by dividing up work. Trends show that CPU's will continue to increase the number of cores. Application needs to embrace parallelism to exploit this change in hardware.

This proposal provides an overview of several parallel versions of algorithms from the C++ standard. Parallel programming has the reputation of being difficult when compared to a counterpart serial program. Additionally, the lack of standardization forces concurrent programmers to often fall back on error-prone, ad-hoc protocols. The standardization of these algorithms will alleviate some of these difficulties and will provide programmers the ability to quickly write parallel programs and gain immediate performance benefits.

Implementation

This proposal describes the common subset of two widely adopted C++ programming libraries. Both Microsoft's Parallel Patterns Library (PPL) and Intel's Threading Building Block Library (TBB), have existing implementations of the algorithms being proposed.

IV. Impact on standard

These changes are entirely based on library extensions and do not require any language features beyond what is available in C++ 11.

V. Design Decisions

The semantics of the algorithms below are based primarily on Microsoft's Parallel Patterns library (PPL) and Intel's Threading Building Blocks library (TBB). One principal design decision is that parallelism is optional. An implementation is free to engage in sequential execution.

Partitioners

In order to parallelize work over multiple cores, the work must be first divided and then distributed over multiple cores. How this work is divided is determined by a partitioner. **`parallel_for`**, **`parallel_for_each`** and **`parallel_transform`** all have overloaded versions in which the partitioner can be specified. This document proposes the following partitioning strategies:

- **`dynamic_partitioner`**: This partitioner is based on any implementation which uses dynamic load balancing and work stealing. Although this partitioner has a greater overhead, it works best with un-balanced workloads.
- **`static_partitioner`**: The default constructor specifies that a parallel loop should divide work into as many chunks as there are cores, and run those chunks in parallel. The constructor that takes a user specified chunk size, divides the work into chunks such that each chunk is no more than the specified chunk size, and runs those chunks in parallel. This partitioner has a lower overhead than the dynamic partitioner as it sacrifices load balancing. This partitioner is best used in situations where the workload is well-balanced, i.e. all iterations in a loop have roughly the same amount of computational complexity.

The `parallel_invoke` Algorithm

Executes multiple (2 or more) callable objects in parallel. **`parallel_invoke`** is a representation of functional parallelism, where the functions are permitted to be invoked concurrently . This algorithm is easy to understand and is useful in cases where the functional blocks are easy to identify. **`parallel_invoke`** only scales up to the number of specified functions (which is usually a small number), unless applied recursively.

It is the programmer's responsibility to determine whether using this function is appropriate and must take into account concurrency issues and race conditions.

Example

```
#include <iostream>
using namespace std;

// Returns the result of adding a value to itself.
int twice (int t) {
   return t + t;
}

int main()
{
   // Define several values.
   int x = 4;
   int y = 10;
   int z = -2;


   // Call the twice function on each value concurrently.
   parallel_invoke(
      [&x] { x = twice(x); },
      [&y] { y = twice(y); },
      [&z] { z = twice(z); }
   );

   // Print the values to the console.
   cout << x << ' ' << y << ' ' << y <<'\n' ;
}
```

This example produces the following output:

```
8 20 -4
```

The `parallel_for` Algorithm

Repeatedly executes the same function in parallel. This algorithm is commonly used to apply a particular function to each element of an array. **`parallel_for`** permits concurrent execution of iterations. Depending on the nature of the workload different partitioning strategies can be used. The **`parallel_for`** algorithm takes the upper and lower bounds of the loop, the optional step value (1 if omitted), and a callable object that is invoked for each iteration of the loop.

Example

```
#include <iostream>
using namespace std;

// Returns the result of adding a value to itself.
int twice(int t) {
   return t + t;
}

int main()
{

   // Call the twice function on each value in the range, concurrently.
   parallel_for(0, 5, [](int i)
   {
        cout << twice(i);
   });

}
```

Because the **parallel_for** algorithm acts on each item in parallel, the order in which the values are printed will vary. This example produces the following output:

```
4 0 6 2 8
```

There are also an additional overload to this function which takes a **partitioner** object, which specifies the partitioning strategy. The three different partitioners that can be used were described above.

parallel_for is not a replacement for the for loop, as often there are cases where it unsafe to use this algorithm. Some of these hazards include races conditions and a variety of performance issues (using locks inside loops, false sharing etc.). Both Microsoft and Intel have published guides on best practices and patterns for this algorithm.

The `parallel_for_each` Algorithm

This algorithm applies a function object on elements in an iterative container (provided by the STL). **parallel_for_each** is semantically the same as **std::for_each** except it is permitted to run concurrently. Like **parallel_for** the order in which the iterations are executed can vary.

Example

```
#include <iostream>
#include <array>
using namespace std;

// Returns the result of adding a value to itself.
int twice(int t) {
   return t + t;
}

int main()
{

   array<int, 4> values = {0, 1, 2, 3};

   // Call the twice function on each value in the range, concurrently.
   parallel_for_each(begin(values), end(values) , [](int i)
   {
         cout << twice(i);
   });

}
```

This example produces the following output:

```
6 0 4 2
```

The `parallel_transform` Algorithm

This algorithm is parallel version of **`std::transform`**. There are two versions of this algorithm, the first performs a unary operation on each of the elements from the input range, in parallel. The second variation performs a binary operation taking one element from the first input range and the other element from the second input range. The return value is a range with the result from each of the function calls. There are also two additional overloads which take a partitioner. The following example illustrates the **`parallel_transform`** function using both the unary and binary function overloads.

Example

```
#include <random>
using namespace std;

// Returns the result of negating the input
int negate_me(int t) {
   return -1 * t;
}

// Returns the result of multiplying the inputs
int multiply(int x, int y) {
   return x * y;
}

int main()
{
    // Create a large vector that contains random integer data.
    vector<int> values1(1250000);
    generate(begin(values1), end(values1), mt19937(42));

    vector<int> values2(1250000);
    generate(begin(values2), end(values2), mt19937(42));

    // Create a vector to hold the results.
    vector<int> results1(values.size());
    vector<int> results2(values.size());

    // Call the function negate_me on each element in values1, and store the results
    in result1.
    parallel_transform(begin(values1), end(values1), begin(results1), [](int n) {
        return negate_me(n);
    });

    // Call the function multiply using an element from values1 as the first parameter
    and one element from values2 as the second parameter.
    parallel_transform(begin(values1), end(values1), begin(values2), begin(results2),
                       [](int n, m) {
        return mulitply(n, m);
    });
}
```

In cases where the function object is concurrently safe, **`parallel_transform`** can be considered to be a faster version of **std::transform**

The `parallel_reduce` Algorithm

This algorithm combines a sequence of elements into one value, in parallel. There are two versions of this function. The first variation computes the sum of all elements in a specified range. The second version similarly computes a specified binary operation (other than sum) on the range. This function is the counterpart to **`std::accumulate`**, however there are two important differences. In the first version the last parameter to **`parallel_reduce`** is the identity whereas the last parameter to std::accumulate is the initial value. In the second version the binary operation given to **`parallel_reduce`** must satisfy the associative property, which is not a requirement for **`std:accumulate`**. Due to these differences and in order to avoid confusion, this function was not called parallel_accumulate.

The following is an example of computing the sum on a given range of data

```
#include <iostream>
using namespace std;

int main()
{

   int data[] = {1,2,3};

   // Computes a sum on the given range, the last input is the identity value for '+'
   auto result = parallel_reduce(begin(data), end(data), 0);

   cout << result; // the output is 7

}
```

The `parallel_sort` Algorithm

This algorithm performs an unstable sort over a sequence of values, possibly in parallel. An unstable sort means that the relative ordering of elements that have the same value may not be preserved. There are two overloads to this algorithm the first takes a begin and end iterator pair. This version sorts the sequence [begin,end) using the **`std::less<T>`** as the default comparator. The second version allows the user to specify the comparator. The following example shows the parallel sorting algorithm by using **`std::greater<int>`** for the comparator.

```
#include <iostream>
#include <random>

using namespace std;

int main()
{
    vector<int> values(25000000);
    generate(begin(values), end(values), mt19937(42));
    parallel_sort(begin(values), end(values), std::greater<int>);
}
```

X **Parallel Algorithms** [parallel_algorithms]

X.1 General [parallel_algorithms.general]

This Clause describe components that C++ programs may use to perform algorithmic operation on containers (Clause 23) and other sequences, possibly in parallel.

X.2 Parallel Algorithms Requirements [parallel_algorithms.requirements]

X.2.2 In general [parallel_algorithms.requirements.general]

1. All algorithms taking a **Partitioner** object can reference one of: **dynamic_partitioner** or **static_partitioner**.
2. When a partitioner is not specified, the default is an implementation based on a dynamic load balancing partitioner.

X.3 **dynamic_partitioner** Class

This partitioner divides the range into runtime-determined chunk sizes as the algorithm runs. It employs a technique for dynamic load balancing.

```
namespace std {
  class dynamic_partitioner {
    public:
       dynamic_partitioner();
       ~dynamic_partitioner();
  }
}
```

X.2 **static_partitioner** Class

This partitioner divides work into a number of chunks that is fixed when the algorithm commences, and runs those chunks in parallel. This partitioner has a lower overhead than the dynamic partitioner.

```
 namespace std {
  class static_partitioner {
    public:
       static_partitioner();
       static_partitioner(size_t chunk_size);
       ~static_partitioner();
  }
}
```

Notes:

- The default constructor divides the range based on the number of hardware thread contexts.
- The constructor that takes a chunk size divides the range into chunks no more than the chunk size specified by the user. This value should only be considered to be a hint.

X.4 Function template **`parallel_invoke`** [parallel_invoke]

Function template that evaluates several callable objects possibly in parallel.

```
template <typename...F>
void parallel_invoke(F...f);
```

Effects:

1. Executes the callable objects supplied as parameters possibly in parallel
2. Function blocks until all inputs have finished executing.

Throws:

1. If one or more of the callable objects throws an exception, one exception is chosen and propagated to the call to **`parallel_invoke`**.

Notes:

1. The number of iterations that complete after an exception is thrown is implementation specific. However, we recommend that all unfinished work be abandoned.
2. Typically the arguments are either function objects, lambda functions or pointers to function. Each argument must a type for which operator() is define.
3. If **`f`** returns a result, the result is ignored.

X.5 Function template **`parallel_for`** [parallel_for]

Function template that may perform parallel iteration over a range of values.

```
template <typename Index, class Function>
void parallel_for(Index first, Index last, Function f);

template <typename Index, class Function>
void parallel_for(Index first, Index last, Index step, Function f);

template <typename Index, class Function, typename Partitioner>
void parallel_for(Index first, Index last, Function f, Partitioner part);

template <typename Index, class Function, typename Partitioner>
void parallel_for(Index first, Index last, Index step, Function f, Partitioner part);
```

Requires:

1. **`Index`** must be signed or unsigned integer type.

Notes:

1. **`parallel_for(first, last, step, f)`** may represent the parallel execution of the loop :
    a. **`for(auto i = first; i < last; i += step) f(i);`**
2. The **`step`** must be > 0, if this requirement is violated, undefined behavior.
3. There is no guarantee that iterations will run in parallel and the order of the iterations are not specified.
4. There shall be no loop carried data dependencies or side-effects across iterations.
5. If **`step`** is omitted, it is implicitly 1.
6. The method returns when the body of the loop has executed for all iterations.

Remarks:

1. In order to disambiguate the second overload from the third overload, the second signature shall not participate in overload resolution if **`decay<Function>::type`** is **`std::partitioner`**.

X.6 Function template **`parallel_for_each`** [parallel_for_each]

Parallel variant of **`std::for_each`**

```
template<class InputIterator, class Function>
void parallel_for_each(InputIterator first, InputIterator last, Function f);

template<class InputIterator, class Function, typename Partitioner>
void parallel_for_each(InputIterator first, InputIterator last, Function f, Partitioner
part);
```

Requires:

1. **`Function`** shall meet the requirements of **`CopyConstructible.`**

Effects:

1. Applies **`f`** to the result of dereferencing every iterator in the range **`[first,last),`** starting from **`first`** and proceeding to **`last-1.`**

Notes:

1. The order of the iterations is not specified

X.6 Function template **`parallel_transform`** [parallel_transform]

Applies a specified function object to each element in a source range, or to a pair of elements from two source ranges, and copies the return values of the function object into a destination range, possibly in parallel. A parallel variation of **`std::transform`**

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator parallel_transform(InputIterator first, InputIterator last,
                                  OutputIterator result, UnaryOperation op);


template<class InputIterator1, class InputIterator2, class OutputIterator,
```

```
        class BinaryOperation >
OutputIterator parallel_transform(InputIterator1 first1, InputIterator1 last1,
                                  InputIterator2 first2, OutputIterator result,
                                  BinaryOperation binary_op);


template<class InputIterator, class OutputIterator, class UnaryOperation,
        class Partitioner>
OutputIterator parallel_transform(InputIterator first, InputIterator last, OutputIterator
                                  result, UnaryOperation op, Partitioner part);


template<class InputIterator1, class InputIterator2, class OutputIterator,
        class BinaryOperation, class Partitioner>
OutputIterator parallel_transform(InputIterator1 first1, InputIterator1 last1,
                                  InputIterator2 first2, OutputIterator result,
                                  BinaryOperation binary_op, Partitioner part);
```

Effects:

1. Assigns through every iterator **i** in the range **[result,result + (last1 - first1))** a new corresponding value equal to **op(*(first1 + (i - result)) or binary_op(*(first1 + (i -result)), *(first2 + (i - result))).**

Requires:

1. **op** and **binary_op** shall not invalidate iterators or subranges, or modify elements in the ranges **[first1,last1], [first2,first2 + (last1 - first1)],** and **[result,result + (last1 - first1)].**

Notes:

1. The overloads which take a **UnaryOperation**, transform the input range into the output range by applying the unary functor to each element in the input range.
2. The overloads which take a **BinaryOperation** transform two input ranges into the output range by applying the binary functor to one element from the first input range as the first parameter, and one element from the second input range as the second parameter.
3. The order of the iterations is not specified

Returns: An output iterator addressing the position one past the final element in the destination range that is receiving the output elements transformed by the function object.

Remarks: In order to disambiguate the second overload from the third overload, the second signature shall not participate in overload resolution if **decay<BinaryFunction>::type** is **std::partitioner**.

X.6 Function template **parallel_reduce** [parallel_reduce]

Computes a reduction over a range.

```
template<typename InputIterator>
const typename iterator_traits<InputIterator>::value_type
    parallel_reduce(InputIterator first, InputIterator last,
                    const typename iterator_traits<ForwardIterator>::value_type i);


template<typename ReduceType, typename InputIterator, typename Function, typename
Reduction>
ReduceType parallel_reduce(InputIterator first, InputIterator last,
                    ReduceType i, Function f, Reduction r);
```

Effects: Computes its result by taking every iterator **i** in the range **[first, last)** and modifying the resulting value.

Requires:

1. **f** must be a function type with the signature **R f(R, R)**, where **R** is the same type as **i** (identity) and the result type of the reduction
2. The first overload requires that the iterator's **value_type, T**, be the same as the identity value type as well as the reduction result type.
3. For the second overload, the identity value type must be the same as the reduction result type, but the iterator's **value_type** may be different from both. The range reduction function **r** is used in the first phase with the identity value as the initial value, and the binary function **f** is applied to sub results in the second phase.

Notes:

1. To perform a parallel reduction, the function divides the range into chunks based on the partitioner. The reduction takes place in two phases, the first phase performs a reduction within each chunk, and the second phase performs a reduction between the partial results from each chunk.

Returns:

1. The result of the reduction

X.6 Function template **parallel_sort** [parallel_sort]

```
template<typename RandomAccessIterator>
void parallel_sort(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, class Compare>
void parallel_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

Effects:

1. Performs an unstable sort of the elements in the range **[first,last)**, possibly in parallel.

Requires:

1. **`RandomAccessIterator`** shall satisfy the requirements of **`ValueSwappable`** (17.6.3.2). The
2. type of **`*first`** shall satisfy the requirements of **`MoveConstructible`** (Table 20) and of **`MoveAssignable`**(Table 22).

Notes:

1. The sort is deterministic (sorting the same sequence will produce the same result every time)
2. The argument **`comp`** is used to determine the relative ordering. The implementer should ensure that it is thread-safe and does not result in any race conditions.
3. When no ordering function is provided, the first overload uses the binary comparator **`std::less`**

Acknowledgements

- Artur Laksberg
- Herb Sutter
- Niklas Gustafsson
- Sana Mithani
- Genevieve Marsh
- Hong Hong
  Rahul V. Patil
  Arch Robison

VII. References

Intel. (n.d.). *Reference Manual*. Retrieved from Intel Threading Building Blocks for Open Source: http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf

Microsoft. (n.d.). *concurrency Namespace*. Retrieved from MSDN: http://msdn.microsoft.com/en-us/library/dd492819.aspx

Microsoft. (n.d.). *Parallel Algorithms*. Retrieved from MSDN: http://msdn.microsoft.com/en-us/library/dd470426.aspx