Parallel Algorithms

Visual Studio 2015

The Parallel Patterns Library (PPL) provides algorithms that concurrently perform work on collections of data. These algorithms resemble those provided by the Standard Template Library (STL).

The parallel algorithms are composed from existing functionality in the Concurrency Runtime. For example, the concurrency::parallel_for algorithm uses a concurrency::structured_task_group object to perform the parallel loop iterations. The **parallel_for** algorithm partitions work in an optimal way given the available number of computing resources.

Sections

The parallel_for Algorithm

The concurrency::parallel_for algorithm repeatedly performs the same task in parallel. Each of these tasks is parameterized by an iteration value. This algorithm is useful when you have a loop body that does not share resources among iterations of that loop.

The **parallel_for** algorithm partitions tasks in an optimum way for parallel execution. It uses a work-stealing algorithm and range stealing to balance these partitions when workloads are unbalanced. When one loop iteration blocks cooperatively, the runtime redistributes the range of iterations that is assigned to the current thread to other threads or processors. Similarly, when a thread completes a range of iterations, the runtime redistributes work from other threads to that thread. The **parallel_for** algorithm also supports *nested parallelism*. When one parallel loop contains another parallel loop, the runtime coordinates processing resources between the loop bodies in an efficient way for parallel execution.

The **parallel_for** algorithm has several overloaded versions. The first version takes a start value, an end value, and a work function (a lambda expression, function object, or function pointer). The second version takes a start value, an end value, a value by which to step, and a work function. The first version of this function uses 1 as the step value. The remaining versions take partitioner objects, which enable you to specify how **parallel_for** should partition ranges among threads. Partitioners are explained in greater detail in the section Partitioning Work in this document.

You can convert many **for** loops to use **parallel_for**. However, the **parallel_for** algorithm differs from the **for** statement in the following ways:

- The **parallel_for** algorithm **parallel_for** does not execute the tasks in a pre-determined order.
- The **parallel_for** algorithm does not support arbitrary termination conditions. The **parallel_for** algorithm stops when the current value of the iteration variable is one less than *_Last*.
- The *_Index_type* type parameter must be an integral type. This integral type can be signed or unsigned.
- The loop iteration must be forward. The **parallel_for** algorithm throws an exception of type std::invalid_argument if the *_Step* parameter is less than 1.
- The exception-handling mechanism for the **parallel_for** algorithm differs from that of a **for** loop. If multiple exceptions occur simultaneously in a parallel loop body, the runtime propagates only one of the exceptions to the thread that called **parallel_for**. In addition, when one loop iteration throws an exception, the runtime does not immediately stop the overall loop. Instead, the loop is placed in the cancelled state and the runtime discards any tasks that have not yet started. For more information about exception-handling and parallel algorithms, see Exception Handling in the Concurrency Runtime.

Although the **parallel_for** algorithm does not support arbitrary termination conditions, you can use cancellation to stop all tasks. For more information about cancellation, see Cancellation in the PPL.

Note

The scheduling cost that results from load balancing and support for features such as cancellation might not overcome the benefits of executing the loop body in parallel, especially when the loop body is relatively small. You can minimize this overhead by using a partitioner in your parallel loop. For more information, see Partitioning Work later in this document.

Example

The following example shows the basic structure of the **parallel_for** algorithm. This example prints to the console each value in the range [1, 5] in parallel.

C++

```cpp
// parallel-for-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <array>
#include <sstream>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
   // Print each value from 1 to 5 in parallel.
   parallel_for(1, 6, [](int value) {
      wstringstream ss;
      ss << value << L' ';
      wcout << ss.str();
   });
}
```

This example produces the following sample output:

```
1 2 4 3 5
```

Because the **parallel_for** algorithm acts on each item in parallel, the order in which the values are printed to the console will vary.

For a complete example that uses the **parallel_for** algorithm, see How to: Write a parallel_for Loop.

[Top]

The parallel_for_each Algorithm

The concurrency::parallel_for_each algorithm performs tasks on an iterative container, such as those provided by the STL, in parallel. It uses the same partitioning logic that the **parallel_for** algorithm uses.

The **parallel_for_each** algorithm resembles the STL std::for_each algorithm, except that the **parallel_for_each** algorithm executes the tasks concurrently. Like other parallel algorithms, **parallel_for_each** does not execute the tasks in a specific order.

Although the **parallel_for_each** algorithm works on both forward iterators and random access iterators, it performs better with random access iterators.

Example

The following example shows the basic structure of the **parallel_for_each** algorithm. This example prints to the console each value in a std::array object in parallel.

C++

```cpp
// parallel-for-each-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <array>
#include <sstream>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
   // Create an array of integer values.
   array<int, 5> values = { 1, 2, 3, 4, 5 };

   // Print each value in the array in parallel.
   parallel_for_each(begin(values), end(values), [](int value) {
      wstringstream ss;
      ss << value << L' ';
      wcout << ss.str();
   });
}
/* Sample output:
   5 4 3 1 2
*/
```

This example produces the following sample output:

```
4 5 1 2 3
```

Because the **parallel_for_each** algorithm acts on each item in parallel, the order in which the values are printed to the console will vary.

For a complete example that uses the **parallel_for_each** algorithm, see How to: Write a parallel_for_each Loop.

[Top]

The parallel_invoke Algorithm

The concurrency::parallel_invoke algorithm executes a set of tasks in parallel. It does not return until each task finishes. This algorithm is useful when you have several independent tasks that you want to execute at the same time.

The **parallel_invoke** algorithm takes as its parameters a series of work functions (lambda functions, function objects, or function pointers). The **parallel_invoke** algorithm is overloaded to take between two and ten parameters. Every function that you pass to **parallel_invoke** must take zero parameters.

Like other parallel algorithms, **parallel_invoke** does not execute the tasks in a specific order. The topic Task Parallelism (Concurrency Runtime) explains how the **parallel_invoke** algorithm relates to tasks and task groups.

Example

The following example shows the basic structure of the **parallel_invoke** algorithm. This example concurrently calls the **twice** function on three local variables and prints the result to the console.

C++

```cpp
// parallel-invoke-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <string>
#include <iostream>

using namespace concurrency;
using namespace std;

// Returns the result of adding a value to itself.
template <typename T>
T twice(const T& t) {
   return t + t;
}

int wmain()
{
   // Define several values.
   int n = 54;
   double d = 5.6;
   wstring s = L"Hello";
```

```
      // Call the twice function on each value concurrently.
      parallel_invoke(
         [&n] { n = twice(n); },
         [&d] { d = twice(d); },
         [&s] { s = twice(s); }
      );

      // Print the values to the console.
      wcout << n << L' ' << d << L' ' << s << endl;
   }
```

This example produces the following output:

```
108 11.2 HelloHello
```

For complete examples that use the **parallel_invoke** algorithm, see How to: Use parallel_invoke to Write a Parallel Sort Routine and How to: Use parallel_invoke to Execute Parallel Operations.

[Top]

The parallel_transform and parallel_reduce Algorithms

The concurrency::parallel_transform and concurrency::parallel_reduce algorithms are parallel versions of the STL algorithms std::transform and std::accumulate, respectively. The Concurrency Runtime versions behave like the STL versions except that the operation order is not determined because they execute in parallel. Use these algorithms when you work with a set that is large enough to get performance and scalability benefits from being processed in parallel.

Important

The **parallel_transform** and **parallel_reduce** algorithms support only random access, bi-directional, and forward iterators because these iterators produce stable memory addresses. Also, these iterators must produce non-**const** l-values.

The parallel_transform Algorithm

You can use the **parallel transform** algorithm to perform many data parallelization operations. For example, you can:

- Adjust the brightness of an image, and perform other image processing operations.

- Sum or take the dot product between two vectors, and perform other numeric calculations on vectors.
- Perform 3-D ray tracing, where each iteration refers to one pixel that must be rendered.

The following example shows the basic structure that is used to call the **parallel_transform** algorithm. This example negates each element of a std::vector object in two ways. The first way uses a lambda expression. The second way uses std::negate, which derives from std::unary_function.

C++

```
// basic-parallel-transform.cpp
// compile with: /EHsc
#include <ppl.h>
#include <random>

using namespace concurrency;
using namespace std;

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

    // Create a vector to hold the results.
    // Depending on your requirements, you can also transform the
    // vector in-place.
    vector<int> results(values.size());

    // Negate each element in parallel.
    parallel_transform(begin(values), end(values), begin(results), [](int n) {
        return -n;
    });

    // Alternatively, use the negate class to perform the operation.
    parallel_transform(begin(values), end(values), begin(values), negate<int>());
}
```

Warning

This example demonstrates the basic use of **parallel_transform**. Because the work function does not perform a significant amount of work, a significant increase in performance is not expected in this example.

The **parallel_transform** algorithm has two overloads. The first overload takes one input range and a unary function. The unary function can be a lambda expression that takes one argument, a function object, or a type that derives from **unary_function**. The second overload takes two input ranges and a binary function. The binary function can be a lambda expression that takes two arguments, a function object, or a type that derives from std::binary_function. The following example illustrates these differences.

C++

```cpp
//
// Demonstrate use of parallel_transform together with a unary function.

// This example uses a lambda expression.
parallel_transform(begin(values), end(values),
    begin(results), [](int n) {
        return -n;
    });

// Alternatively, use the negate class:
parallel_transform(begin(values), end(values),
    begin(results), negate<int>());

//
// Demonstrate use of parallel_transform together with a binary function.

// This example uses a lambda expression.
parallel_transform(begin(values), end(values), begin(results),
    begin(results), [](int n, int m) {
        return n * m;
    });

// Alternatively, use the multiplies class:
parallel_transform(begin(values), end(values), begin(results),
    begin(results), multiplies<int>());
```

Important

The iterator that you supply for the output of **parallel_transform** must completely overlap the input iterator or not overlap at all. The behavior of this algorithm is unspecified if the input and output iterators partially overlap.

The parallel_reduce Algorithm

The **parallel_reduce** algorithm is useful when you have a sequence of operations that satisfy the associative property. (This algorithm does not require the commutative property.) Here are some of the operations that you can perform with **parallel_reduce**:

- Multiply sequences of matrices to produce a matrix.
- Multiply a vector by a sequence of matrices to produce a vector.
- Compute the length of a sequence of strings.
- Combine a list of elements, such as strings, into one element.

The following basic example shows how to use the **parallel_reduce** algorithm to combine a sequence of strings into one string. As with the examples for **parallel_transform**, performance gains are not expected in this basic example.

C++

```
// basic-parallel-reduce.cpp
// compile with: /EHsc
#include <ppl.h>
#include <iostream>
#include <string>
#include <vector>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create a vector of strings.
    vector<wstring> words;
    words.push_back(L"Lorem ");
    words.push_back(L"ipsum ");
    words.push_back(L"dolor ");
    words.push_back(L"sit ");
    words.push_back(L"amet, ");
    words.push_back(L"consectetur ");
    words.push_back(L"adipiscing ");
    words.push_back(L"elit.");

    // Reduce the vector to one string in parallel.
    wcout << parallel_reduce(begin(words), end(words), wstring()) << endl;
}

/* Output:
   Lorem ipsum dolor sit amet, consectetur adipiscing elit.
*/
```

In many cases, you can think of **parallel_reduce** as shorthand for the use of the **parallel_for_each** algorithm together with the concurrency::combinable class.

Example: Performing Map and Reduce in Parallel

A *map* operation applies a function to each value in a sequence. A *reduce* operation combines the elements of a sequence into one value. You can use the Standard Template Library (STL) std::transform std::accumulate classes to perform map and reduce operations. However, for many problems, you can use the **parallel_transform** algorithm to perform the map operation in parallel and the **parallel_reduce** algorithm perform the reduce operation in parallel.

The following example compares the time that it takes to compute the sum of prime numbers serially and in parallel. The map phase transforms non-prime values to 0 and the reduce phase sums the values.

C++

```
// parallel-map-reduce-sum-of-primes.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <array>
#include <numeric>
#include <iostream>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
   __int64 begin = GetTickCount();
   f();
   return GetTickCount() - begin;
}

// Determines whether the input value is prime.
bool is_prime(int n)
{
   if (n < 2)
      return false;
   for (int i = 2; i < n; ++i)
   {
      if ((n % i) == 0)
         return false;
   }
   return true;
}
```

```
int wmain()
{
   // Create an array object that contains 200000 integers.
   array<int, 200000> a;

   // Initialize the array such that a[i] == i.
   iota(begin(a), end(a), 0);

   int prime_sum;
   __int64 elapsed;

   // Compute the sum of the numbers in the array that are prime.
   elapsed = time_call([&] {
       transform(begin(a), end(a), begin(a), [](int i) {
         return is_prime(i) ? i : 0;
      });
      prime_sum = accumulate(begin(a), end(a), 0);
   });
   wcout << prime_sum << endl;
   wcout << L"serial time: " << elapsed << L" ms" << endl << endl;

   // Now perform the same task in parallel.
   elapsed = time_call([&] {
      parallel_transform(begin(a), end(a), begin(a), [](int i) {
         return is_prime(i) ? i : 0;
      });
      prime_sum = parallel_reduce(begin(a), end(a), 0);
   });
   wcout << prime_sum << endl;
   wcout << L"parallel time: " << elapsed << L" ms" << endl << endl;
}
/* Sample output:
   1709600813
   serial time: 7406 ms

   1709600813
   parallel time: 1969 ms
*/
```

For another example that performs a map and reduce operation in parallel, see How to: Perform Map and Reduce Operations in Parallel.

[Top]

Partitioning Work

To parallelize an operation on a data source, an essential step is to *partition* the source into multiple sections that can be accessed concurrently by multiple threads. A partitioner specifies how a parallel algorithm should partition ranges among threads. As explained previously in this document, the PPL uses a default partitioning mechanism that creates an initial workload and then uses a work-stealing algorithm and range stealing to balance these partitions when workloads are unbalanced. For example, when one loop iteration completes a range of iterations, the runtime redistributes work from other threads to that thread. However, for some scenarios, you might want to specify a different partitioning mechanism that is better suited to your problem.

The **parallel_for**, **parallel_for_each**, and **parallel_transform** algorithms provide overloaded versions that take an additional parameter, *_Partitioner*. This parameter defines the partitioner type that divides work. Here are the kinds of partitioners that the PPL defines:

concurrency::affinity_partitioner
: Divides work into a fixed number of ranges (typically the number of worker threads that are available to work on the loop). This partitioner type resembles **static_partitioner**, but improves cache affinity by the way it maps ranges to worker threads. This partitioner type can improve performance when a loop is executed over the same data set multiple times (such as a loop within a loop) and the data fits in cache. This partitioner does not fully participate in cancellation. It also does not use cooperative blocking semantics and therefore cannot be used with parallel loops that have a forward dependency.

concurrency::auto_partitioner
: Divides work into an initial number of ranges (typically the number of worker threads that are available to work on the loop). The runtime uses this type by default when you do not call an overloaded parallel algorithm that takes a *_Partitioner* parameter. Each range can be divided into sub-ranges, and thereby enables load balancing to occur. When a range of work completes, the runtime redistributes sub-ranges of work from other threads to that thread. Use this partitioner if your workload does not fall under one of the other categories or you require full support for cancellation or cooperative blocking.

concurrency::simple_partitioner
: Divides work into ranges such that each range has at least the number of iterations that are specified by the given chunk size. This partitioner type participates in load balancing; however, the runtime does not divide ranges into sub-ranges. For each worker, the runtime checks for cancellation and performs load-balancing after *_Chunk_size* iterations complete.

concurrency::static_partitioner
: Divides work into a fixed number of ranges (typically the number of worker threads that are available to work on the loop). This partitioner type can improve performance because it does not use work-stealing and therefore has less overhead. Use this partitioner type when each iteration of a parallel loop performs a fixed and uniform amount of work and you do not require support for cancellation or forward cooperative blocking.

Warning

The **parallel_for_each** and **parallel_transform** algorithms support only containers that use random access iterators (such as std::vector) for the static, simple, and affinity partitioners. The use of containers that use bidirectional and forward iterators produces a compile-time error. The default partitioner, **auto_partitioner**, supports all three of these iterator types.

Typically, these partitioners are used in the same way, except for **affinity_partitioner**. Most partitioner types do not maintain state and are not modified by the runtime. Therefore you can create these partitioner objects at the call site, as shown in the following example.

C++

```
// static-partitioner.cpp
// compile with: /EHsc
#include <ppl.h>

using namespace concurrency;

void DoWork(int n)
{
    // TODO: Perform a fixed amount of work...
}

int wmain()
{
    // Use a static partitioner to perform a fixed amount of parallel work.
    parallel_for(0, 100000, [](int n) {
        DoWork(n);
    }, static_partitioner());
}
```

However, you must pass an **affinity_partitioner** object as a non-**const**, l-value reference so that the algorithm can store state for future loops to reuse. The following example shows a basic application that performs the same operation on a data set in parallel multiple times. The use of **affinity_partitioner** can improve performance because the array is likely to fit in cache.

C++

```
// affinity-partitioner.cpp
// compile with: /EHsc
#include <ppl.h>
#include <array>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create an array and fill it with zeroes.
    array<unsigned char, 8 * 1024> data;
    data.fill(0);
```

```
        // Use an affinity partitioner to perform parallel work on data
        // that is likely to remain in cache.
        // We use the same affinitiy partitioner throughout so that the
        // runtime can schedule work to occur at the same location for each
        // iteration of the outer loop.

        affinity_partitioner ap;
        for (int i = 0; i < 100000; i++)
        {
            parallel_for_each(begin(data), end(data), [](unsigned char& c)
            {
                c++;
            }, ap);
        }
    }
```

Caution

Use caution when you modify existing code that relies on cooperative blocking semantics to use **static_partitioner** or **affinity_partitioner**. These partitioner types do not use load balancing or range stealing, and therefore can alter the behavior of your application.

The best way to determine whether to use a partitioner in any given scenario is to experiment and measure how long it takes operations to complete under representative loads and computer configurations. For example, static partitioning might provide significant speedup on a multi-core computer that has only a few cores, but it might result in slowdowns on computers that have relatively many cores.

[Top]

Parallel Sorting

The PPL provides three sorting algorithms: concurrency::parallel_sort, concurrency::parallel_buffered_sort, and concurrency::parallel_radixsort. These sorting algorithms are useful when you have a data set that can benefit from being sorted in parallel. In particular, sorting in parallel is useful when you have a large dataset or when you use a computationally-expensive compare operation to sort your data. Each of these algorithms sorts elements in place.

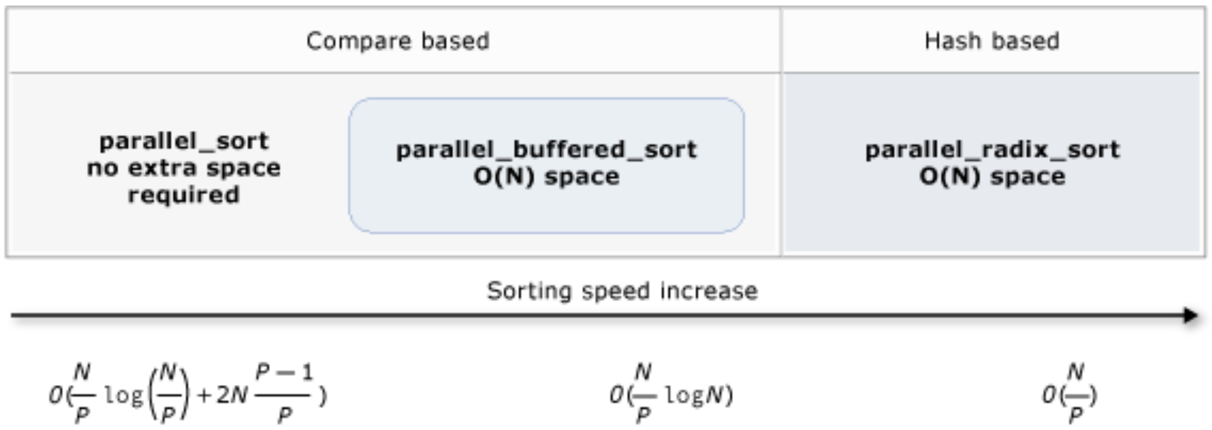
The **parallel_sort** and **parallel_buffered_sort** algorithms are both compare-based algorithms. That is, they compare elements by value. The **parallel_sort** algorithm has no additional memory requirements, and is suitable for general-purpose sorting. The **parallel_buffered_sort** algorithm can perform better than **parallel_sort**, but it requires *O(N)* space.

The **parallel_radixsort** algorithm is hash-based. That is, it uses integer keys to sort elements. By using keys, this algorithm can directly compute the destination of an element instead of using comparisons. Like **parallel_buffered_sort**, this algorithm requires *O(N)* space.

The following table summarizes the important properties of the three parallel sorting algorithms.

Algorithm	Description	Sorting mechanism	Sort Stability	Memory requirements	Time Complexity	Iterator access
parallel_sort	General-purpose compare-based sort.	Compare-based (ascending)	Unstable	None	*O((N/P)log(N/P) + 2N((P-1)/P))*	Random
parallel_buffered_sort	Faster general-purpose compare-based sort that requires O(N) space.	Compare-based (ascending)	Unstable	Requires additional *O(N)* space	*O((N/P)log(N))*	Random
parallel_radixsort	Integer key-based sort that requires O(N) space.	Hash-based	Stable	Requires additional *O(N)* space	*O(N/P)*	Random

The following illustration shows the important properties of the three parallel sorting algorithms more graphically.


Compare based
Hash based
parallel_sort no extra space required
parallel_buffered_sort O(N) space
parallel_radix_sort O(N) space
Sorting speed increase
$O(\frac{N}{P}\log(\frac{N}{P})+2N\frac{P-1}{P})$
$O(\frac{N}{P}\log N)$
$O(\frac{N}{P})$

These parallel sorting algorithms follow the rules of cancellation and exception handling. For more information about cancellation and exception handling in the Concurrency Runtime, see Canceling Parallel Algorithms and Exception Handling in the Concurrency Runtime.

Tip

These parallel sorting algorithms support move semantics. You can define a move assignment operator to enable swap operations to occur more efficiently. For more information about move semantics and the move assignment operator, see Rvalue Reference Declarator: &&, and Move Constructors and Move Assignment Operators (C++). If you do not provide a move assignment operator or swap function, the sorting algorithms use the copy constructor.

The following basic example shows how to use **parallel_sort** to sort a **vector** of **int** values. By default, **parallel_sort** uses std::less to compare values.

C++

```cpp
// basic-parallel-sort.cpp
// compile with: /EHsc
#include <ppl.h>
#include <random>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create and sort a large vector of random values.
    vector<int> values(25000000);
    generate(begin(values), end(values), mt19937(42));
    parallel_sort(begin(values), end(values));

    // Print a few values.
    wcout << "V(0)        = " << values[0] << endl;
    wcout << "V(12500000) = " << values[12500000] << endl;
    wcout << "V(24999999) = " << values[24999999] << endl;
}
/* Output:
   V(0)        = -2147483129
   V(12500000) = -427327
   V(24999999) = 2147483311
*/
```

This example shows how to provide a custom compare function. It uses the std::complex::real method to sort std::complex<double> values in ascending order.

C++

```cpp
// For this example, ensure that you add the following #include directive:
// #include <complex>
// Create and sort a large vector of random values.
vector<complex<double>> values(25000000);
generate(begin(values), end(values), mt19937(42));
parallel_sort(begin(values), end(values),
    [](const complex<double>& left, const complex<double>& right) {
        return left.real() < right.real();
    });

// Print a few values.
wcout << "V(0)        = " << values[0] << endl;
wcout << "V(12500000) = " << values[12500000] << endl;
wcout << "V(24999999) = " << values[24999999] << endl;
/* Output:
   V(0)        = (383,0)
   V(12500000) = (2.1479e+009,0)
   V(24999999) = (4.29497e+009,0)
*/
```

This example shows how to provide a hash function to the **parallel_radixsort** algorithm. This example sorts 3-D points. The points are sorted based on their distance from a reference point.

C++

```
// parallel-sort-points.cpp
// compile with: /EHsc
#include <ppl.h>
#include <random>
#include <iostream>

using namespace concurrency;
using namespace std;

// Defines a 3-D point.
struct Point
{
    int X;
    int Y;
    int Z;
};

// Computes the Euclidean distance between two points.
size_t euclidean_distance(const Point& p1, const Point& p2)
{
    int dx = p1.X - p2.X;
    int dy = p1.Y - p2.Y;
    int dz = p1.Z - p2.Z;
    return static_cast<size_t>(sqrt((dx*dx) + (dy*dy) + (dz*dz)));
}
```

```
int wmain()
{
    // The central point of reference.
    const Point center = { 3, 2, 7 };

    // Create a few random Point values.
    vector<Point> values(7);
    mt19937 random(42);
    generate(begin(values), end(values), [&random] {
        Point p = { random()%10, random()%10, random()%10 };
        return p;
    });

    // Print the values before sorting them.
    wcout << "Before sorting:" << endl;
    for_each(begin(values), end(values), [center](const Point& p) {
        wcout << L'(' << p.X << L"," << p.Y << L"," << p.Z
              << L") D = " << euclidean_distance(p, center) << endl;
    });
    wcout << endl;

    // Sort the values based on their distances from the reference point.
    parallel_radixsort(begin(values), end(values),
        [center](const Point& p) -> size_t {
            return euclidean_distance(p, center);
        });

    // Print the values after sorting them.
    wcout << "After sorting:" << endl;
    for_each(begin(values), end(values), [center](const Point& p) {
        wcout << L'(' << p.X << L"," << p.Y << L"," << p.Z
              << L") D = " << euclidean_distance(p, center) << endl;
    });
    wcout << endl;
}
/* Output:
    Before sorting:
    (2,7,6) D = 5
    (4,6,5) D = 4
    (0,4,0) D = 7
    (3,8,4) D = 6
    (0,4,1) D = 7
    (2,5,5) D = 3
    (7,6,9) D = 6

    After sorting:
    (2,5,5) D = 3
    (4,6,5) D = 4
    (2,7,6) D = 5
    (3,8,4) D = 6
    (7,6,9) D = 6
    (0,4,0) D = 7
    (0,4,1) D = 7
*/
```

For illustration, this example uses a relatively small data set. You can increase the initial size of the vector to experiment with performance improvements over larger sets of data.

This example uses a lambda expression as the hash function. You can also use one of the built-in implementations of the std::hash class or define your own specialization. You can also use a custom hash function object, as shown in this example:

C++

```cpp
// Functor class for computing the distance between points.
class hash_distance
{
public:
    hash_distance(const Point& reference)
        : m_reference(reference)
    {
    }

    size_t operator()(const Point& pt) const {
        return euclidean_distance(pt, m_reference);
    }

private:
    Point m_reference;
};
```

C++

```cpp
// Use hash_distance to compute the distance between points.
parallel_radixsort(begin(values), end(values), hash_distance(center));
```

The hash function must return an integral type (std::is_integral::value must be **true**). This integral type must be convertible to type **size_t**.

Choosing a Sorting Algorithm

In many cases, **parallel_sort** provides the best balance of speed and memory performance. However, as you increase the size of your data set, the number of available processors, or the complexity of your compare function, **parallel_buffered_sort** or **parallel_radixsort** can perform better. The best way to determine which sorting algorithm to use in any given scenario is to experiment and measure how long it takes to sort typical data under representative computer configurations. Keep the following guidelines in mind when you choose a sorting strategy.

- The size of your data set. In this document, a *small* dataset contains fewer than 1,000 elements, a *medium* dataset contains between 10,000 and 100,000 elements, and a *large* dataset contains more than 100,000 elements.
- The amount of work that your compare function or hash function performs.
- The amount of available computing resources.
- The characteristics of your data set. For example, one algorithm might perform well for data that is already nearly sorted, but not as well for data that is completely unsorted.
- The chunk size. The optional *_Chunk_size* argument specifies when the algorithm switches from a parallel to a serial sort implementation as it subdivides the overall sort into smaller units of work. For example, if you provide 512, the algorithm switches to serial implementation when a unit of work contains 512 or fewer elements. A serial implementation can improve overall performance because it eliminates the overhead that is required to process data in parallel.

It might not be worthwhile to sort a small dataset in parallel, even when you have a large number of available computing resources or your compare function or hash function performs a relatively large amount of work. You can use std::sort function to sort small datasets. (**parallel_sort** and **parallel_buffered_sort** call **sort** when you specify a chunk size that is larger than the dataset; however, **parallel_buffered_sort** would have to allocate *O(N)* space, which could take additional time due to lock contention or memory allocation.)

If you must conserve memory or your memory allocator is subject to lock contention, use **parallel_sort** to sort a medium-sized dataset. **parallel_sort** requires no additional space; the other algorithms require *O(N)* space.

Use **parallel_buffered_sort** to sort medium-sized datasets and when your application meets the additional *O(N)* space requirement. **parallel_buffered_sort** can be especially useful when you have a large number of computing resources or an expensive compare function or hash function.

Use **parallel_radixsort** to sort large datasets and when your application meets the additional *O(N)* space requirement. **parallel_radixsort** can be especially useful when the equivalent compare operation is more expensive or when both operations are expensive.

Caution

Implementing a good hash function requires that you know the dataset range and how each element in the dataset is transformed to a corresponding unsigned value. Because the hash operation works on unsigned values, consider a different sorting strategy if unsigned hash values cannot be produced.

The following example compares the performance of **sort**, **parallel_sort**, **parallel_buffered_sort**, and **parallel_radixsort** against the same large set of random data.

C++

```
// choosing-parallel-sort.cpp
// compile with: /EHsc
#include <ppl.h>
#include <random>
#include <iostream>
#include <windows.h>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
   __int64 begin = GetTickCount();
   f();
   return GetTickCount() - begin;
}

const size_t DATASET_SIZE = 10000000;

// Create
// Creates the dataset for this example. Each call
// produces the same predefined sequence of random data.
vector<size_t> GetData()
{
    vector<size_t> data(DATASET_SIZE);
    generate(begin(data), end(data), mt19937(42));
    return data;
}

int wmain()
{
    // Use std::sort to sort the data.
    auto data = GetData();
    wcout << L"Testing std::sort...";
    auto elapsed = time_call([&data] { sort(begin(data), end(data)); });
    wcout << L" took " << elapsed << L" ms." <<endl;

    // Use concurrency::parallel_sort to sort the data.
    data = GetData();
    wcout << L"Testing concurrency::parallel_sort...";
    elapsed = time_call([&data] { parallel_sort(begin(data), end(data)); });
    wcout << L" took " << elapsed << L" ms." <<endl;
```

```
        // Use concurrency::parallel_buffered_sort to sort the data.
        data = GetData();
        wcout << L"Testing concurrency::parallel_buffered_sort...";
        elapsed = time_call([&data] { parallel_buffered_sort(begin(data), end(data)); });
        wcout << L" took " << elapsed << L" ms." <<endl;

        // Use concurrency::parallel_radixsort to sort the data.
        data = GetData();
        wcout << L"Testing concurrency::parallel_radixsort...";
        elapsed = time_call([&data] { parallel_radixsort(begin(data), end(data)); });
        wcout << L" took " << elapsed << L" ms." <<endl;
    }
    /* Sample output (on a computer that has four cores):
        Testing std::sort... took 2906 ms.
        Testing concurrency::parallel_sort... took 2234 ms.
        Testing concurrency::parallel_buffered_sort... took 1782 ms.
        Testing concurrency::parallel_radixsort... took 907 ms.
    */
```

In this example, which assumes that it is acceptable to allocate *O(N)* space during the sort, **parallel_radixsort** performs the best on this dataset on this computer configuration.

[Top]

Related Topics

Title	Description
How to: Write a parallel_for Loop	Shows how to use the **parallel_for** algorithm to perform matrix multiplication.
How to: Write a parallel_for_each Loop	Shows how to use the **parallel_for_each** algorithm to compute the count of prime numbers in a std::array object in parallel.
How to: Use parallel_invoke to Write a Parallel Sort Routine	Shows how to use the **parallel_invoke** algorithm to improve the performance of the bitonic sort algorithm.
How to: Use parallel_invoke to Execute Parallel Operations	Shows how to use the **parallel_invoke** algorithm to improve the performance of a program that performs multiple operations on a shared data source.

How to: Perform Map and Reduce Operations in Parallel	Shows how to use the **parallel_transform** and **parallel_reduce** algorithms to perform a map and reduce operation that counts the occurrences of words in files.
Parallel Patterns Library (PPL)	Describes the PPL, which provides an imperative programming model that promotes scalability and ease-of-use for developing concurrent applications.
Cancellation in the PPL	Explains the role of cancellation in the PPL, how to cancel parallel work, and how to determine when a task group is canceled.
Exception Handling in the Concurrency Runtime	Explains the role of exception handling in the Concurrency Runtime.

Reference

parallel_for Function

parallel_for_each Function

parallel_invoke Function

affinity_partitioner Class

auto_partitioner Class

simple_partitioner Class

static_partitioner Class

parallel_sort Function

parallel_buffered_sort Function

parallel_radixsort Function