Using C++17 Parallel Algorithms for Better Performance

Billy

September 11th, 2018

This post is part of a regular series of posts where the C++ product team here at Microsoft and other guests answer questions we have received from customers. The questions can be about anything C++ related: MSVC toolset, the standard language and library, the C++ standards committee, isocpp.org, CppCon, etc. Today's post is by Billy O'Neal.

C++17 added support for parallel algorithms to the standard library, to help programs take advantage of parallel execution for improved performance. MSVC first added experimental support for some algorithms in 15.5, and the experimental tag was removed in 15.7.

The interface described in the standard for the parallel algorithms doesn't say exactly how a given workload is to be parallelized. In particular, the interface is intended to express parallelism in a general form that works for heterogeneous machines, allowing SIMD parallelism like that exposed by SSE, AVX, or NEON, vector "lanes" like that exposed in GPU programming models, and traditional threaded parallelism.

Our parallel algorithms implementation currently relies entirely on library support, not on special support from the compiler. This means our implementation will work with any tool currently consuming our standard library, not just MSVC's compiler. In particular, we test that it works with Clang/LLVM and the version of EDG that powers Intellisense.

How to: Use Parallel Algorithms

To use the parallel algorithms library, you can follow these steps:

1. Find an algorithm call you wish to optimize with parallelism in your program. Good candidates are algorithms which do more than O(n) work like sort, and show up as taking reasonable amounts of time when profiling your application.
2. Verify that code you supply to the algorithm is safe to parallelize.
3. Choose a parallel execution policy. (Execution policies are described below.)
4. If you aren't already, `#include <execution>` to make the parallel execution policies available.
5. Add one of the execution policies as the first parameter to the algorithm call to parallelize.
6. Benchmark the result to ensure the parallel version is an improvement. Parallelizing is not always faster, particularly for non-random-access iterators, or when the input size is small, or when the additional parallelism creates contention on external resources like a disk.

For the sake of example, here's a program we want to make faster. It times how long it takes to sort a million doubles.

```cpp
// compile with:
//  debug: cl /EHsc /W4 /WX /std:c++latest /Fedebug /MDd .\program.cpp
//  release: cl /EHsc /W4 /WX /std:c++latest /Ferelease /MD /O2 .\program.cpp
#include <stddef.h>
#include <stdio.h>
#include <algorithm>
#include <chrono>
#include <random>
#include <ratio>
#include <vector>

using std::chrono::duration;
using std::chrono::duration_cast;
using std::chrono::high_resolution_clock;
using std::milli;
using std::random_device;
using std::sort;
using std::vector;

const size_t testSize = 1'000'000;
const int iterationCount = 5;

void print_results(const char *const tag, const vector<double>& sorted,
                   high_resolution_clock::time_point startTime,
                   high_resolution_clock::time_point endTime) {
  printf("%s: Lowest: %g Highest: %g Time: %fms\n", tag, sorted.front(),
         sorted.back(),
         duration_cast<duration<double, milli>>(endTime - startTime).count());
}

int main() {
  random_device rd;

  // generate some random doubles:
  printf("Testing with %zu doubles...\n", testSize);
  vector<double> doubles(testSize);
  for (auto& d : doubles) {
    d = static_cast<double>(rd());
  }

  // time how long it takes to sort them:
  for (int i = 0; i < iterationCount; ++i)
  {
    vector<double> sorted(doubles);
    const auto startTime = high_resolution_clock::now();
    sort(sorted.begin(), sorted.end());
    const auto endTime = high_resolution_clock::now();
    print_results("Serial", sorted, startTime, endTime);
  }
}
```

Parallel algorithms depend on available hardware parallelism, so ensure you test on hardware whose performance you care about. You don’t need a lot of cores to show wins, and many parallel algorithms are divide and conquer problems that won’t show perfect scaling with thread count anyway, but more is still better. For the purposes of this example, we tested on an Intel 7980XE system with 18 cores and 36 threads. In that test, debug and release builds of this program produced the following output:

[text]
.\debug.exe
Testing with 1000000 doubles...
Serial: Lowest: 1349 Highest: 4.29497e+09 Time: 310.176500ms
Serial: Lowest: 1349 Highest: 4.29497e+09 Time: 304.714800ms
Serial: Lowest: 1349 Highest: 4.29497e+09 Time: 310.345800ms
Serial: Lowest: 1349 Highest: 4.29497e+09 Time: 303.302200ms
Serial: Lowest: 1349 Highest: 4.29497e+09 Time: 290.694300ms

C:\Users\bion\Desktop>.\release.exe
Testing with 1000000 doubles...
Serial: Lowest: 2173 Highest: 4.29497e+09 Time: 74.590400ms
Serial: Lowest: 2173 Highest: 4.29497e+09 Time: 75.703500ms
Serial: Lowest: 2173 Highest: 4.29497e+09 Time: 87.839700ms
Serial: Lowest: 2173 Highest: 4.29497e+09 Time: 73.822300ms
Serial: Lowest: 2173 Highest: 4.29497e+09 Time: 73.757400ms
[/text]

Next, we need to ensure our sort call is safe to parallelize. Algorithms are safe to parallelize if the “element access functions” — that is, iterator operations, predicates, and anything else you ask the algorithm to do on your behalf follow the normal “any number of readers or at most one writer” rules for data races. Moreover, they must not

throw exceptions (or throw exceptions rarely enough that terminating the program if they do throw is OK).

Next, choose an execution policy. Currently, the standard includes the parallel policy, denoted by `std::execution::par`, and the parallel unsequenced policy, denoted by `std::execution::par_unseq`. In addition to the requirements exposed by the parallel policy, the parallel unsequenced policy requires that your element access functions tolerate weaker than concurrent forward progress guarantees. That means that they don't take locks or otherwise perform operations that require threads to concurrently execute to make progress. For example, if a parallel algorithm runs on a GPU and tries to take a spinlock, the thread spinning on the spinlock may prevent other threads on the GPU from ever executing, meaning the spinlock may never be unlocked by the thread holding it, deadlocking the program. You can read more about the nitty gritty requirements in the [algorithms.parallel.defns] and [algorithms.parallel.exec] sections of the C++ standard. If in doubt, use the parallel policy. In this example, we are using the built-in double less-than operator which doesn't take any locks, and an iterator type provided by the standard library, so we can use the parallel unsequenced policy.

Note that the Visual C++ implementation implements the parallel and parallel unsequenced policies the same way, so you should not expect better performance for using par_unseq on our implementation, but implementations may exist that can use that additional freedom someday.

In the doubles sort example above, we can now add

```
1 #include <execution>
```

to the top of our program. Since we're using the parallel unsequenced policy, we add `std::execution::par_unseq` to the algorithm call site. (If we were using the parallel policy, we would use `std::execution::par` instead.) With this change the for loop in main becomes the following:

```
1  for (int i = 0; i < iterationCount; ++i)
2  {
3    vector<double> sorted(doubles);
4    const auto startTime = high_resolution_clock::now();
5    // same sort call as above, but with par_unseq:
6    sort(std::execution::par_unseq, sorted.begin(), sorted.end());
7    const auto endTime = high_resolution_clock::now();
8    // in our output, note that these are the parallel results:
9    print_results("Parallel", sorted, startTime, endTime);
10 }
```

Last, we benchmark:

[text]
.\debug.exe
Testing with 1000000 doubles...
Parallel: Lowest: 6642 Highest: 4.29496e+09 Time: 54.815300ms
Parallel: Lowest: 6642 Highest: 4.29496e+09 Time: 49.613700ms
Parallel: Lowest: 6642 Highest: 4.29496e+09 Time: 49.504200ms
Parallel: Lowest: 6642 Highest: 4.29496e+09 Time: 49.194200ms
Parallel: Lowest: 6642 Highest: 4.29496e+09 Time: 49.162200ms

.\release.exe
Testing with 1000000 doubles...
Parallel: Lowest: 18889 Highest: 4.29496e+09 Time: 20.971100ms
Parallel: Lowest: 18889 Highest: 4.29496e+09 Time: 17.510700ms
Parallel: Lowest: 18889 Highest: 4.29496e+09 Time: 17.823800ms
Parallel: Lowest: 18889 Highest: 4.29496e+09 Time: 20.230400ms
Parallel: Lowest: 18889 Highest: 4.29496e+09 Time: 19.461900ms
[/text]

The result is that the program is faster for this input. How you benchmark your program will depend on your own success criteria. Parallelization does have some overhead and will be slower than the serial version if N is small enough, depending on memory and cache effects, and other factors specific to your particular workload. In this example, if I

set N to 1000, the parallel and serial versions run at approximately the same speed, and if I change N to 100, the serial version is 10 times faster. Parallelization can deliver huge wins but choosing where to apply it is important.

Current Limitations of the MSVC Implementation of Parallel Algorithms

We built the parallel `reverse`, and it was 1.6x slower than the serial version on our test hardware, even for large values of N. We also tested with another parallel algorithms implementation, HPX, and got similar results. That doesn't mean it was wrong for the standards committee to add those to the STL; it just means the hardware our implementation targets didn't see improvements. As a result we provide the signatures for, but do not actually parallelize, algorithms which merely permute, copy, or move elements in sequential order. If we get feedback with an example where parallelism would be faster, we will look into parallelizing these. The affected algorithms are:

- copy
- copy_n
- fill
- fill_n
- move
- reverse
- reverse_copy
- rotate
- rotate_copy
- swap_ranges

Some algorithms are unimplemented at this time and will be completed in a future release. The algorithms we parallelize in Visual Studio 2017 15.8 are:

- adjacent_difference
- adjacent_find
- all_of
- any_of
- count
- count_if
- equal
- exclusive_scan
- find
- find_end
- find_first_of
- find_if
- for_each
- for_each_n
- inclusive_scan
- mismatch
- none_of
- partition
- reduce
- remove
- remove_if
- search
- search_n
- sort
- stable_sort
- transform
- transform_exclusive_scan
- transform_inclusive_scan
- transform_reduce

Parsing Code for MSVC's Parallel Algorithms Implementation

Design Goals for MSVC's Parallel Algorithms Implementation

While the standard specifies the interface of the parallel algorithms library, it doesn't say at all how algorithms should be parallelized, or even on what hardware they should be parallelized. Some implementations of C++ may parallelize by using GPUs or other heterogeneous compute hardware if available on the target. `copy` doesn't make sense for our implementation to parallelize, but it does make sense on an implementation that targets a GPU or similar accelerator. We value the following aspects in our implementation:

Composition with platform locks

Microsoft previously shipped a parallelism framework, ConcRT, which powered parts of the standard library. ConcRT allows disparate workloads to transparently use the hardware available, and lets threads complete each other's work, which can increase overall throughput. Basically, whenever a thread would normally go to sleep running a ConcRT workload, it suspends the chore it's currently executing and runs other ready-to-run chores instead. This non-blocking behavior reduces context switches and can produce higher overall throughput than the Windows threadpool our parallel algorithms implementation uses. However, it also means that ConcRT workloads do not compose with operating system synchronization primitives like `SRWLOCK`, NT events, semaphores, COM single threaded apartments, window procedures, etc. We believe that is an unacceptable trade-off for the "by default" implementation in the standard library.

The standard's parallel unsequenced policy allows a user to declare that they support the kinds of limitations lightweight user-mode scheduling frameworks like ConcRT have, so we may look at providing ConcRT-like behavior in the future. At the moment however, we only have plans to make use of the parallel policy. If you can meet the requirements, you should use the parallel unsequenced policy anyway, as that may lead to improved performance on other implementations, or in the future.

Usable performance in debug builds

We care about debugging performance. Solutions that require the optimizer to be turned on to be practical aren't suitable for use in the standard library. If I add a `Concurrency::parallel_sort` call to the previous example program, ConcRT's parallel sort a bit faster in release but almost 100 times slower in debug:

```
1 for (int i = 0; i < iterationCount; ++i)
2 {
3   vector<double> sorted(doubles);
4   const auto startTime = high_resolution_clock::now();
5   Concurrency::parallel_sort(sorted.begin(), sorted.end());
6   const auto endTime = high_resolution_clock::now();
7   print_results("ConcRT", sorted, startTime, endTime);
8 }
```

[text]
C:\Users\bion\Desktop>.\debug.exe
Testing with 1000000 doubles...
ConcRT: Lowest: 5564 Highest: 4.29497e+09 Time: 23910.081300ms
ConcRT: Lowest: 5564 Highest: 4.29497e+09 Time: 24096.297700ms
ConcRT: Lowest: 5564 Highest: 4.29497e+09 Time: 23868.098500ms
ConcRT: Lowest: 5564 Highest: 4.29497e+09 Time: 24159.756200ms
ConcRT: Lowest: 5564 Highest: 4.29497e+09 Time: 24950.541500ms

C:\Users\bion\Desktop>.\release.exe
Testing with 1000000 doubles...
ConcRT: Lowest: 1394 Highest: 4.29496e+09 Time: 19.019000ms
ConcRT: Lowest: 1394 Highest: 4.29496e+09 Time: 16.348000ms
ConcRT: Lowest: 1394 Highest: 4.29496e+09 Time: 15.699400ms
ConcRT: Lowest: 1394 Highest: 4.29496e+09 Time: 15.907100ms
ConcRT: Lowest: 1394 Highest: 4.29496e+09 Time: 15.859500ms
[/text]

Composition with other programs on the system and parallelism frameworks

frameworks

Scheduling in our implementation is handled by the Windows system thread pool. The thread pool takes advantage of information not available to the standard library, such as what other threads on the system are doing, what kernel resources threads are waiting for, and similar.
It chooses when to create more threads, and when to terminate them. It's also shared with other system components, including those not using C++.

For more information about the kinds of optimizations the thread pool does on your (and our) behalf, check out Pedro Teixeira's talk on the thread pool, as well as official documentation for the `CreateThreadpoolWork`, `SubmitThreadpoolWork`, `WaitForThreadpoolWorkCallbacks`, and `CloseThreadpoolWork` functions.

Above all, parallelism is an optimization

If we can't come up with a practical benchmark where the parallel algorithm wins for reasonable values of N, it will not be parallelized. We view being twice as fast at N=1'000'000'000 and 3 orders of magnitude slower when N=100 as an unacceptable tradeoff. If you want "parallelism no matter what the cost" there are plenty of other implementations which work with MSVC, including HPX and Threading Building Blocks.

Similarly, the C++ standard permits parallel algorithms to allocate memory, and throw `std::bad_alloc` when they can't acquire memory. In our implementation we fall back to a serial version of the algorithm if additional resources can't be acquired.

Benchmark parallel algorithms and speed up your own applications

Algorithms that take more than O(n) time (like sort) and are called with larger N than 2,000 are good places to consider applying parallelism. We want to make sure this feature behaves as you expect; please give it a try. If you have any feedback or suggestions for us, let us know. We can be reached via the comments below, via email (visualcpp@microsoft.com) and you can provide feedback via Help > Report A Problem in the product, or via Developer Community. You can also find us on Twitter (@VisualC) and Facebook (msftvisualcpp).

This article was updated on 2018-10-31 to indicate that partition was parallelized in 15.8.

Billy O'Neal

Follow Billy

Posted in Documentation, General C++ Series

Read next

Exploring Clang Tooling, Part 0: Building Your Code with Clang

This post is part of a regular series of posts where the C++ product team and other guests answer questions we have received from customers. The questions can be about an

Stephen Kelly September 18, 2018

Parallel Custom Build Tools in Visual Studio 2017

Many projects need to use additional tools during their build to generate sources for further compilation or perform other custom build tasks. VC projects have always sup

olgaark September 18, 2018

1 comment 0 comment

5 comments

Log in to join the discussion.

sal p April 17, 2019 10:14 am

hi
i was trying to test out the c++17 parallel algorithms with visual studio 2019 . The headers are included but the algorithms are not recognized such as sort, for_each or reduce…
The <execution> header file does not give ANY error but when the functions are added it does not recognize the parallel algorithms with std::execution::par or any other options.
I tried to change the language options from default to C++17 and then to std::latest but still the same error.
Any idea why this happens..I have not used visual studio 2017 but all the post i see refer to 2017 version working std::latest flag.
I appreciate if you can help. thanks..

Log in to Reply

Billy O'Neal April 17, 2019 8:02 pm

As far as I am aware there are no observable changes to how this works in VS2019; the algorithms should be available under both /std:c++17 and /std:c++latest. Are you sure that setting is taking effect in the place you're expecting?

Log in to Reply

Tim Bodin July 7, 2019 7:12 pm

I'm having the same problem as sal p; VS19 compile err msg says "no instance of overloaded function sort matches the argument list". It is hanging up on the revised sort line….the one that adds par_unseq as the first parameter. The (non-parallel) version without that runs just fine. In case you might ask, I have the #include <execution> at the top, and have moved it around, without any complaint from the (pre-) compiler.

Log in to Reply

Serge Rogatch August 18, 2019 7:42 pm

Please, see here how "copy" operation benefits from parallelization: https://stackoverflow.com/a/44948720/1915854 . Let me know if you have questions.

Log in to Reply

Billy O'Neal August 19, 2019 6:55 pm

That's a bugfix that memcpy doesn't use the right instruction cocktail for Ryzen, not that std::copy(par should do anything.

Log in to Reply

Relevant Links

Getting Started with C++ in VS
Bring Your Existing C++ Code to VS
C++ Code Editing & Navigation
C++ Unit Testing
C++ Debugging & Diagnosics
Collaborating with Your Team in VS
C++ Windows Development
C++ pnux Development
C++ Android & iOS Development
C++ Game Development

Topics

Uncategorized
Announcement
New Feature
CMake
Visual Studio Code
Linux
Vcpkg
General C++ Series
OpenFolder
performance
Experimental

Archive

October 2019
September 2019
August 2019
July 2019
June 2019
May 2019
April 2019
March 2019
February 2019
January 2019
December 2018

Stay informed