



 fedand 24 января 2018 в 11:34

Parallel STL. Быстрый способ ускорить C++ STL код

Автор оригинала: V. Polin, M. Dvorskiy

Блог компании Intel, Высокая производительность, Программирование, C++, Параллельное программирование

Перевод

За пару последних десятилетий, пока вычислительные системы эволюционировали от одноядерных скалярных до многоядерных векторных архитектур, значительно выросла популярность управляемых языков, а также появились новые языки программирования. Но старый добрый C++, позволяющий писать высокопроизводительный код, остается более чем популярным. Однако, до недавнего времени стандарт языка не предоставлял каких-либо инструментов для выражения параллелизма. Новая версия стандарта (C++17 [1]) предоставляет набор параллельных алгоритмов Parallel STL, дающий возможность преобразовать существующий последовательный C++ код в параллельный, что, в свою очередь, позволяет задействовать такие аппаратные возможности, как многопоточность и векторизация. Эта статья познакомит вас с основами Parallel STL и его реализацией в Intel Parallel Studio XE 2018.



Введение в Parallel STL

Итак, судьба поддержки параллелизма в C++ складывалась очень непросто. На рисунке ниже можно познакомиться с имеющимися для этого разноуровневыми средствами в разных версиях C++, в том числе и целым «зоопарком» различных «внешних» средств разработки параллельного ПО, созданных производителями ПО или «железа».

	C++11/14	C++17	Будущее C++	Другие подходы
Верхний уровень (Сообщения, события, Flow Graph)	<code>std::async</code> , <code>std::future</code>	<code>std::async</code> , <code>std::future</code>	resumable functions	MPI*, Microsoft AAL*, Intel® TBB flow graph, Qualcomm Symphony*, etc
Средний уровень (системные потоки)	<code>std::thread</code> + manual sync	Parallel STL (<code>par</code> , <code>par_unseq</code>)	task block, <code>for_loop</code>	OpenMP*, Intel TBB, Cilk*, Nvidia Thrust*, Qualcomm Symphony*, OpenCL*, Microsoft PPL*, OpenACC*, etc
Нижний уровень (векторизация)	—————	Parallel STL (<code>par_unseq</code>)	Parallel STL (<code>unseq</code> , <code>vec</code>) <code>for_loop</code>	intrinsics, auto-vectorization, Intel® Cilk™ Plus, OpenCL*, Nvidia CUDA*, OpenMP* 4, OpenACC* etc

Рисунок 1. Эволюция параллелизма в C++

Parallel STL – это расширение стандартной библиотеки шаблонов C++ (STL – Standard Template Library), в котором вводится ключевое понятие «политики выполнения» (execution policy).

Политика выполнения – это C++ класс, используемый в качестве уникального типа для перегрузки алгоритмов STL. Для удобства использования стандарт также определяет по одному объекту каждого такого класса, который может быть передан как аргумент при вызове алгоритмов. Они могут использоваться как с хорошо известными алгоритмами (`transform`, `for_each`, `copy_if`), так и с новыми алгоритмами, появившимися в C++17 (`reduce`, `transform_reduce`, `inclusive_scan` и т. д.). Необходимо уточнить, что не все алгоритмы в C++17 поддерживают политики.

Далее под параллельным выполнением алгоритма будем понимать выполнение на нескольких ядрах CPU; под векторизацией будем понимать выполнение с задействованием векторных регистров процессора. Поддержка параллельных политик разрабатывалась в течение нескольких лет как «Technical Specification for C++ Extensions for Parallelism*» (Parallelism TS). Теперь эта спецификация вошла в стандарт языка C++17. Поддержка векторных политик может войти во вторую версию спецификации Parallelism TS (n4698 [2], p0076 [3]). В целом, эти документы описывают 5 различных политик выполнения (Рис. 2):

- **sequenced_policy (seq)** говорит о том, что алгоритм может выполняться последовательно [1].
- **parallel_policy (par)** указывает на то, что алгоритм может выполняться параллельно [1]. Пользовательские функции, вызываемые во время работы алгоритма, не должны порождать «гонку данных».
- **parallel_unsequenced_policy (par_unseq)** указывает на то, что алгоритм может выполняться параллельно и векторно [1].
- **unsequenced_policy (unseq)** – класс, являющийся частью черновика Parallelism TS v2 [2] и показывающий, что алгоритм может выполняться векторно. Эта политика требует, чтобы все пользовательские функции, функторы, передаваемые в алгоритм в качестве параметров, не препятствовали векторизации (не содержали зависимостей по данным, не вызывали «гонки данных», и т.д.).
- **vector_policy (vec)** (также из [2]) говорит о том, что алгоритм может выполняться векторно, но в отличие от **unseq** политика **vec** гарантирует обработку данных в том порядке, в каком они бы обрабатывались при последовательном выполнении (сохраняет ассоциативность).

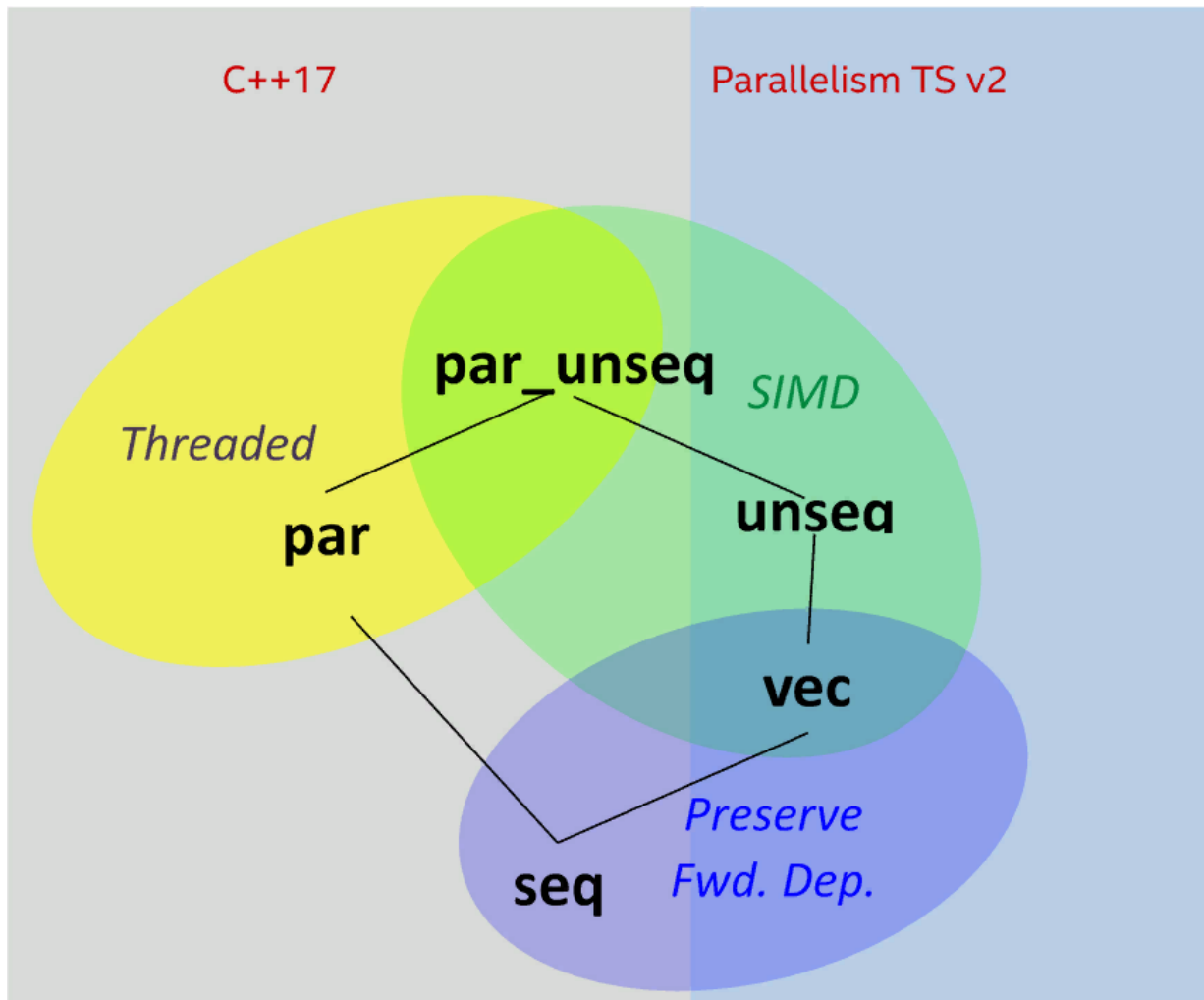


Рисунок 2. Политики выполнения для алгоритмов

Рисунок выше демонстрирует отношения между этими политиками. Чем выше политика на схеме, тем большую степень свободы (с точки зрения параллелизма) она допускает.

Ниже приведены примеры использования алгоритмов STL и Parallel STL согласно стандарту C++17:

```
#include <execution>
#include <algorithm>

void increment_seq( float *in, float *out, int N ) {
    using namespace std;
    transform( in, in + N, out, []( float f ) {
        return f+1;
    });
}

void increment_unseq( float *in, float *out, int N ) {
    using namespace std;
    using namespace std::execution;
    transform( unseq, in, in + N, out, []( float f ) {
        return f+1;
    });
}

void increment_par( float *in, float *out, int N ) {
    using namespace std;
    using namespace std::execution;
    transform( par, in, in + N, out, []( float f ) {
        return f+1;
    });
}
```

Где запись

```
std::transform( in, in + N, out, foo );
```

эквивалентна следующему циклу:

```
for (x = in; x < in+N; ++x) *(out+(x-in)) = foo(x);
```

и

```
std::transform( unseq, in, in + N, out, foo );
```

можно представить в виде следующего цикла (наша реализация использует `#pragma omp simd` на нижнем уровне, другие реализации Parallel STL могут использовать другие способы реализации политики `unseq`)

```
#pragma omp simd
for (x = in; x < in+N; ++x) *(out+(x-in)) = foo(x);
```

и

```
std::transform( par, in, in + N, out);
```

можно выразить следующим образом:

```
tbb::parallel_for (in, in+N, [=] (x) {
    *(out+(x-in)) = foo(x);
});
```

Обзор реализации Parallel STL в Intel® Parallel Studio XE 2018

Реализация Parallel STL от Intel – часть Intel® Parallel Studio XE 2018. Мы предлагаем переносимую реализацию алгоритмов, которые могут выполняться как параллельно, так и векторно. Реализация оптимизирована и оттестирована для процессоров Intel®. Для работы с политиками `par` и `par_unseq` мы используем Intel® Threading Building Blocks (Intel® TBB), для политик `unseq`, `par_unseq` – векторизацию средствами OpenMP*. Политика `vec` в Intel Parallel Studio XE 2018 не представлена.

После установки Parallel STL вам нужно установить переменные окружения, как описано в этом документе. Там же есть актуальный список алгоритмов, которые имеют параллельную и/или векторную реализации. Для остальных алгоритмов политики выполнения также применимы, но при этом будет вызываться последовательная версия.

Для достижения наилучших результатов с нашей реализацией Parallel STL мы рекомендуем использовать компилятор Intel® C++ Compiler 2018. Но вы можете пользоваться и другими компиляторами, поддерживающими C++11. Чтобы получить положительный эффект от векторизации, компилятор должен также поддерживать OpenMP 4.0 (`#pragma omp simd`). Для использования политик `par`, `par_unseq` необходима библиотека Intel TBB.

Чтобы добавить Parallel STL в свое приложение, выполните следующие шаги:

1. Добавьте `#include «pstl/execution»`. Затем одну или нескольких строк в зависимости от того, какие алгоритмы вы собираетесь использовать:

```
#include «pstl/algorithm»
#include «pstl/numeric»
#include «pstl/memory»
```

Заметьте, что следует писать не просто `#include <execution>`, а `#include «pstl/execution»`. Это сделано специально, чтобы избежать конфликтов с заголовочными файлами стандартной библиотеки C++.

- По месту использования алгоритмов и политик укажите пространство имен `std` и `pstl::execution` соответственно.
- Скомпилируйте код с опцией поддержки C++11 или выше. Используйте подходящую опцию компиляции для включения OpenMP-векторизации (например, для Intel C++ Compiler `-qopenmp-simd` (`/Qopenmp-simd` для Windows*)).
- Для получения лучшей производительности, укажите целевую платформу. Для Intel® C++ Compiler используйте подходящие опции из списка: `-xHOST`, `-xCORE-AVX2`, `-xMIC-AVX512` для Linux* или `/QxHOST`, `/QxCORE-AVX2`, `/QxMIC-AVX512` для Windows.
- Ссылкуйте с Intel TBB. На Windows это произойдет автоматически; на других платформах добавьте `-ltbb` к опциям компоновщика.

Intel Parallel Studio XE 2018 содержит примеры использования Parallel STL, которые вы можете собрать и запустить. Их можно скачать [здесь](#).

Эффективная векторизация, параллелизм и совместимость при использовании нашей реализации Parallel STL

В теории, Parallel STL разрабатывался как интуитивно понятный способ для C++ разработчиков писать программы для параллельных вычислительных систем с общей памятью. Рассмотрим подход, при котором теория коррелирует с лучшими практиками распараллеливания вложенных циклов, например, такой подход, как «Векторизуйте внутренний уровень, распараллеливайте внешний» («Vectorize Innermost, Parallelize Outermost» [VIPO]) [4]. В качестве примера рассмотрим гамма-коррекцию изображения – нелинейную операцию, используемую для изменения яркости каждого пикселя изображения. Заметим, что мы должны отключить автовекторизацию для последовательного выполнения алгоритмов, чтобы показать разницу между последовательным и векторным выполнением. (Иным образом это различие можно увидеть только с компиляторами, не поддерживающими автоматическую векторизацию, но поддерживающими OpenMP-векторизацию)

Рассмотрим пример последовательного выполнения:

```
#include <algorithm>
void ApplyGamma(Image& rows, float g) {
    using namespace std;
    for_each(rows.begin(), rows.end(), [g](Row &r) {
        transform(r.cbegin(), r.cend(), r.begin(),
            [g](float v) {
                return pow(v, g);
            });
    });
}
```

Функция `ApplyGamma` получает по ссылке изображение, представленное в виде набора строк, и вызывает `std::for_each` для итерирования строк. Лямбда-функция, вызываемая для каждой строки, выполняет цикл по пикселям с помощью `std::transform` для изменения яркости каждого пикселя.

Как было описано ранее, Parallel STL предоставляет параллельную и векторную версии алгоритмов `for_each` и `transform`. Другими словами, политика, передаваемая в качестве первого аргумента в алгоритм приводит к выполнению параллельной и/или векторной версии этого алгоритма.

Возвращаясь к примеру выше, можно заметить, что все вычисления производятся в лямбда-функции, которая вызывается из алгоритма `transform`. Так давайте попробуем «убить двух зайцев одним выстрелом» и перепишем пример, используя политику `par_unseq`:

```
void ApplyGamma(Image& rows, float g) {
    using namespace pstl::execution;
    std::for_each(rows.begin(), rows.end(), [g](Row &r) {
        // Inner parallelization and vectorization
        std::transform(par_unseq, r.cbegin(), r.cend(), r.begin(),
            [g](float v) {
```

```

        return pow(v, g);
    });
});
}

```

Производительность гамма-коррекции (Parallel STL)
на Intel(R) Xeon(R) CPU E3-1240 v5 @ 3.50GHz (4C/8T)
Размер изображения: 800x600

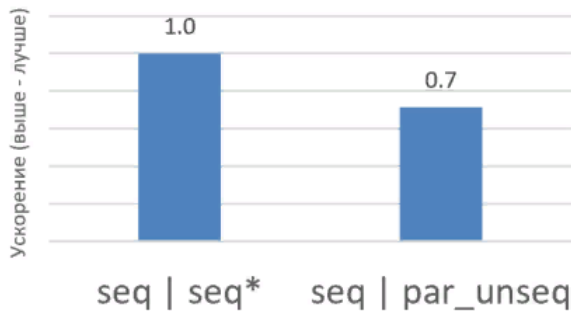


Рисунок 3. **Par_unseq** для внутреннего цикла

Удивительно, но чуда не произошло (Рис. 3). Производительность с политикой **par_unseq** хуже, чем при последовательном выполнении. Это отличный пример того, как НЕ нужно использовать Parallel STL. Если вы будете профилировать код с помощью, например, Intel® VTune Amplifier XE, вы можете увидеть много кэш-промахов, вызванных тем, что потоки, работающие на разных ядрах, обращаются к одним и тем же кэш-линиям (такой эффект известен как «ложное разделение ресурсов» [false sharing]).

Как отмечалось ранее, Parallel STL помогает нам выразить параллелизм среднего (с помощью системных потоков) и нижнего (с помощью векторизации) уровней параллелизма. В общем случае, чтобы получить наибольшее ускорение, оцените время выполнения алгоритма и сравните его с накладными расходами на параллелизм и векторизацию. Мы рекомендуем, чтобы время последовательного выполнения было хотя бы в 2 раза больше, чем накладные расходы на каждом уровне параллелизма. Помимо этого:

- Распараллеливайте самый верхний уровень; ищите максимальное количество работы для выполнения в параллельном режиме.
- Если это дает вам достаточную эффективность распараллеливания – цель достигнута. Если нет – распараллеливайте уровень ниже.
- Убедитесь, что алгоритм эффективно использует кэш.
- Пробуйте векторизовать самый нижний уровень. Старайтесь уменьшать количество условных переходов в векторизуемой функции и соблюдать равномерность доступа к памяти.
- Больше рекомендаций ищите в [4].

Рекомендации предполагают, что правильное использование параллельных и векторных политик на разных уровнях может дать более высокую производительность, а именно:

```

void ApplyGamma(Image& rows, float g) {
    using namespace pstl::execution;
    // Outer parallelization
    std::for_each(par, rows.begin(), rows.end(), [g](Row &r) {
        // Inner vectorization
        std::transform(unseq, r.cbegin(), r.cend(), r.begin(),
            [g](float v) {
                return pow(v, g);
            });
    });
}

```

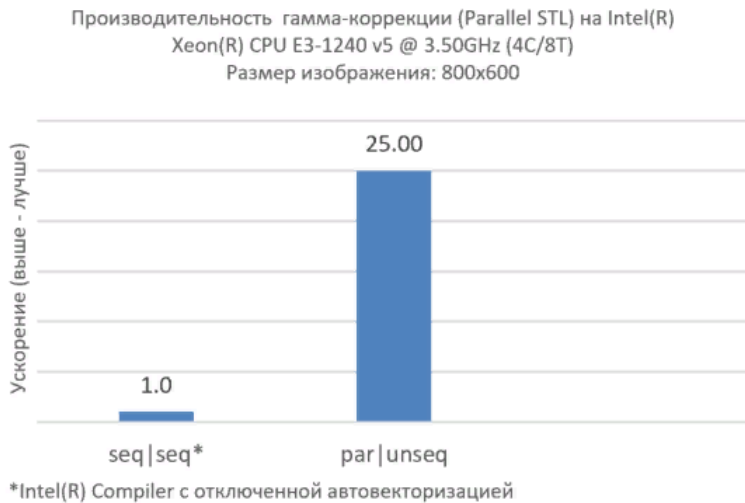



Рисунок 4. Векторизация на внутреннем уровне, параллельность на внешнем

Теперь мы получили эффективную параллельную обработку одного изображения (Рис. 4), но реальные приложения, как правило, обрабатывают множество изображений (Рис. 5). Параллелизм на более высоком уровне может плохо работать со стандартными алгоритмами. В этом случае мы предлагаем использовать Parallel STL совместно с Intel TBB.

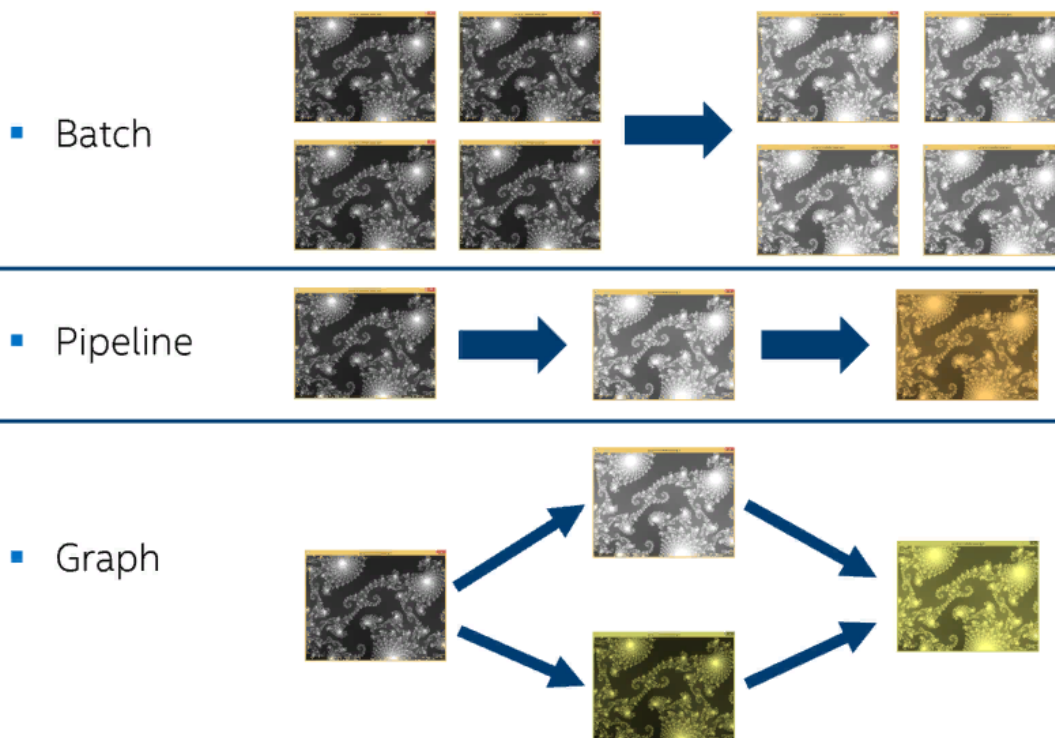


Рисунок 5. Способы обработки множества изображений

Это позволяет применить задачи (tasks) или параллельные конструкции (например, вычислительный граф [flow graph], конвейер [pipeline]) Intel TBB на самом верхнем уровне и алгоритмы Parallel STL на более низких уровнях, не заботясь о создании избыточного количества логических потоков в системе. Пример:

```
void Function() {
    Image img1, img2;
    // Prepare img1 and img2
    tbb::parallel_invoke(
        [&img1] { img1.ApplyGamma(gamma1); },
        [&img2] { img2.ApplyGamma(gamma2); }
    );
}
```

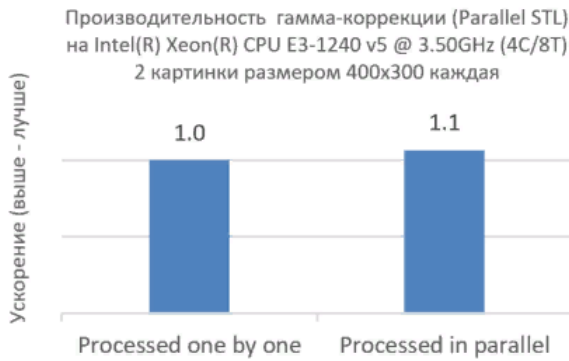


Рисунок 6. Совместное использование Intel TBB и Parallel STL

Как показано на рисунке 6, одновременная обработка двух изображений с помощью Intel TBB не уменьшает производительность, а напротив, даже немного увеличивает ее. Это показывает, что выражение параллелизма на уровне ниже и на самом низком уровне максимально эффективно использует ядра CPU.

А теперь рассмотрим ситуацию, когда у нас есть больше изображений для обработки и больше ядер CPU.

```
tbb::parallel_for(images.begin(), images.end(),
    [](image* img) {applyGamma(img->rows(), 1.1);}
);
```

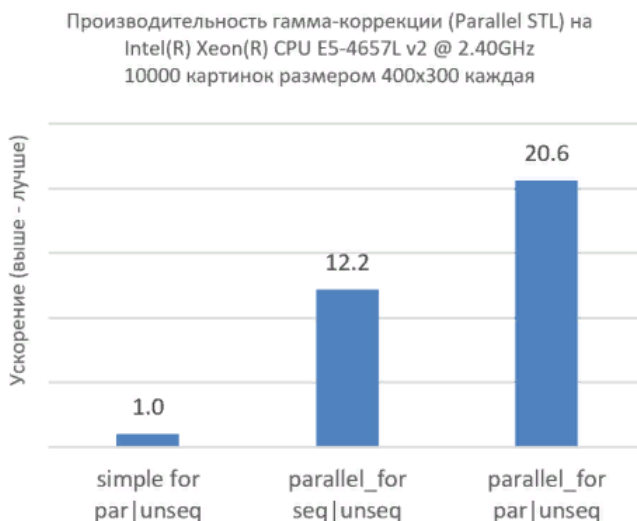


Рисунок 7. Совместное использование Intel TBB и Parallel STL на большем числе изображений и большем числе ядер CPU.

На рисунке выше видно, что одновременная обработка множества изображений с помощью Intel TBB (*parallel_for*) резко увеличивает производительность. В самом деле, взгляните на первый столбец, где мы последовательно пробегаем все изображения, при этом каждое изображение обрабатывается параллельно на более низком уровне. Добавление только лишь параллелизма на самом верхнем уровне (*parallel_for*) без параллелизма на уровне ниже (**par**) существенно увеличивает производительность, но этого недостаточно, чтобы наиболее полно использовать ресурсы ядер CPU. Третий столбец показывает, что параллелизм на всех уровнях резко увеличивает производительность. Это показывает эффективность совместного использования Intel TBB и нашей реализации Parallel STL.

Заключение

Parallel STL – значительный шаг в эволюции C++ параллелизма, что делает его легко применимым к алгоритмам стандартной библиотеки STL как при модернизации кода, так и при создании новых приложений. Эта часть стандарта добавляет к языку C++ возможности векторизации и параллелизма без использования нестандартных или самописных расширений, а политики выполнения предоставляют контроль над использованием таких возможностей, абстрагируясь от «железа». Parallel STL позволяет разработчикам сфокусироваться на выражении параллелизма в их приложениях, не заботясь о низкоуровневом управлении потоками и векторными регистрами. В дополнение к эффективной высокопроизводительной реализации высокоуровневых алгоритмов наша реализация Parallel STL демонстрирует эффективное совместное использование с параллельными паттернами Intel TBB. Однако Parallel STL не панацея. Параллельные и векторные политики нужно

использовать обдуманно, в зависимости от размерности задачи, типа и объема данных, а также от кода функций, функторов, используемых в алгоритмах. Для достижения высокой производительности можно порекомендовать некоторые способы, описанные в [4].

Вы можете найти актуальные версии Parallel STL и Intel TBB, а также дополнительную информацию на следующих сайтах:

- Открытая реализация Parallel STL на github
- Официальный сайт Intel TBB
- Intel TBB на github
- Документация Intel TBB

Нам нужны ваши отзывы. А оставить их вы можете здесь:

- Страница проекта на github
- Форум Intel TBB

Ссылки:

[1] ISO/IEC 14882:2017, Programming Language C++

[2] n4698, Working Draft, Technical Specification for C++ Extensions for Parallelism Version 2, Programming Language C++ (WG21)

[3] P0076r4, Vector and Wavefront Policies, Programming Language C++ (WG21)

[4] Robert Geva, Code Modernization Best Practices: Multi-level Parallelism for Intel® Xeon® and Intel® Xeon Phi™ Processors, IDF15 – Webcast

* прочие названия и бренды могут быть объявлены интеллектуальной собственностью других лиц

Теги: intel компиляторы оптимизация, векторизация, c++, TBB, параллельное программирование

Хэбы: Блог компании Intel, Высокая производительность, Программирование, C++, Параллельное программирование

↑ +22 ↓ 78 👁 14,6k 💬 7 ➦ Поделиться



Андрей @fedand

Пользователь



Intel

Компания

Сайт Twitter ВКонтакте

ПОХОЖИЕ ПУБЛИКАЦИИ

17 октября 2011 в 19:26

«Слабо загрузить 40 ядер?» или простой конкурс параллельного программирования Acceler8 2011

↑ +29 👁 16,4k 📖 28 💬 31

22 апреля 2011 в 20:13

Начинаем конкурс параллельного программирования Threading Challenge


↑ +37 👁 32,8k 📖 25 💬 65

16 декабря 2009 в 14:42

Параллельное программирование в черном ящике

↑ +15 👁 9,2k 📖 10 💬 47

Комментарии 7

 **Alcor** 24 января 2018 в 15:33    -1 

А можно ещё добавить чисто C-вариант? По моему опыту, параллелизация приводит к хорошей утилизации ресурсов всех ядер CPU, но добавление новых слоев абстракции зачастую съедает часть выгоды.

В моем случае отказ от STL/OpenMP в пользу C-массивов/Thread позволил отказаться от 60-ядерного сервера в пользу i5 с сохранением производительности.

 **fedand** 24 января 2018 в 16:30      0 

Parallel STL — часть C++ стандарта. Теоретически эти алгоритмы можно использовать и с C-массивами. Например,


```
int* in;
int* out;
// Выделить память, заполнить in
std::copy(pstl::execution::unseq, in, in+n, out);
```

Такой пример должен отработать. Изначально в качестве параллельного движка мы использовали TBB, но теоретически дизайн позволяет и другие "бэкенды" использовать.

OpenMP здесь используется только для векторизации, не для многопоточности.

 **Alcor** 24 января 2018 в 16:46      -1 

Нет, к сожалению, под рукой компилятора с поддержкой pstl, но на том, который есть (gcc 4.8.5), однопоточный вариант std::copy медленнее тетсру на треть. Потому и было интересно сравнение с классическим C.

 **fedand** 24 января 2018 в 16:53      0 


Однопоточный вариант std::copy с политиками или обычный, без политик, из стандартной библиотеки?

 **Alcor** 24 января 2018 в 17:03      0 

Обычный

 **fedand** 24 января 2018 в 17:28      0 

Да, такое возможно. Таких чистых C-вариантов алгоритмов Parallel STL не предоставляет

 **Grisha_Kirilin** 24 января 2018 в 19:16      +3 

Для более или менее свежих компиляторов это не так:

Quick C++ Benchmark

Support Quick Bench ▾ More ▾

```

1 #include<vector>
2 #include<algorithm>
3 #include <cstring>
4
5
6 static void using_std_copy(benchmark::State& stat
7     size_t size = 1 << 16;
8     std::vector<float> source( size );
9     std::vector<float> target( size );
10
11     for (auto _ : state) {
12         std::copy( source.begin(), source.end(), targ
13     }
14 }
15 // Register the function as a benchmark
16 BENCHMARK(using_std_copy);
17
18 static void using_memcpy(benchmark::State& state)
19     size_t size = 1 << 16;
20     std::vector<float> source( size );
21     std::vector<float> target( size );
22
23     for (auto _ : state) {
24         std::memcpy( target.data(), source.data(), si
25     }
26 }
27 }
28 BENCHMARK(using_memcpy);
29

```

compiler = gcc-5.5 ▾

std = c++17 ▾

optim = O3 ▾

Run Benchmark

☒ Record disassembly

☐ Clear cached results

using_std_copy

using_memcpy

ratio (CPU time / Noop time)

☐ Show Noop bar

using_std_copy

using_memcpy

Можете попробовать разные варианты сами.

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки Неделя Месяц

Коронавирус: опасная иллюзия неопасности

↑ +37 👁 18,7k 📖 38 💬 58

Что нового в Ubuntu 20.04

↑ +68 👁 50,9k 📖 69 💬 142

Знаменитые дизайнеры vs научные исследования про читаемость шрифтов

↑ +109 👁 15,2k 📖 80 💬 53

Как научиться разработке на Python: новый видеокурс Яндекса

↑ +42 👁 15k 📖 422 💬 25

Изготовление мини ПК на APU Ryzen или компьютер дальнбойщика

↑ +72 👁 25,1k 📖 52 💬 74

Ваш аккаунт	Разделы	Информация	Услуги
Войти	Публикации	Устройство сайта	Реклама
Регистрация	Новости	Для авторов	Тарифы
	Хабы	Для компаний	Контент
	Компании	Документы	Семинары
	Пользователи	Соглашение	Мегапроекты
	Песочница	Конфиденциальность	

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.