Структурированное параллельное программирование и Intel Threading Building Blocks

Anton Potapov, 2016

Structured Parallel Programming: Patterns for Efficient Computation

– Michael McCool

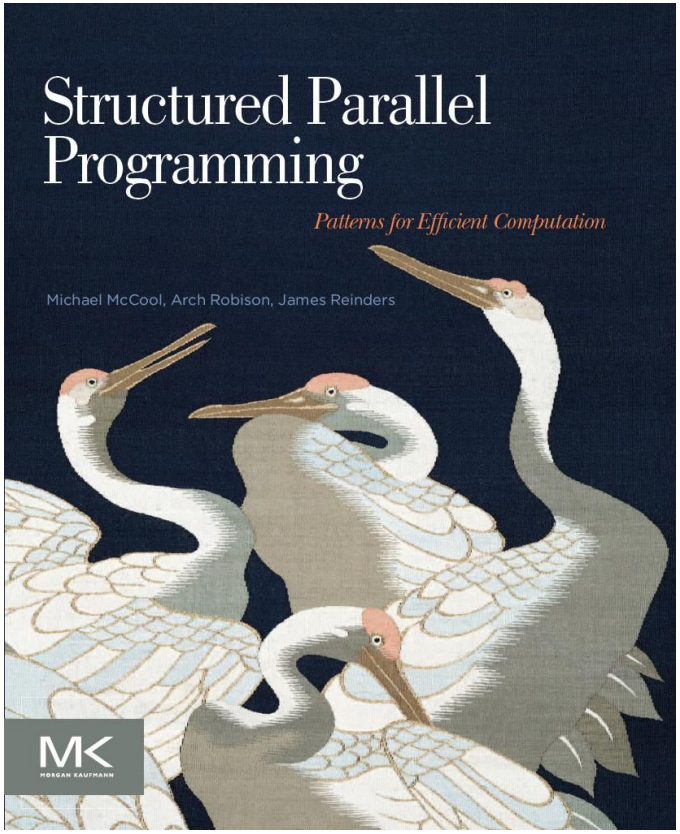
– Arch Robison

– James Reinders

• Uses Cilk Plus and TBB as primary frameworks for examples.

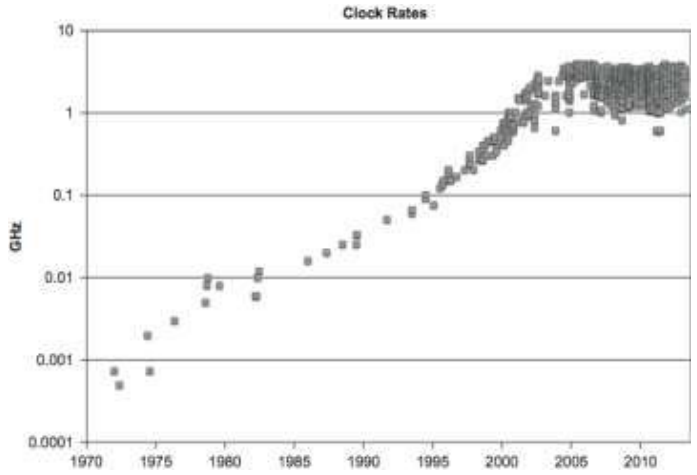
• Appendices concisely summarize Cilk Plus and TBB.

• www.parallelbook.com

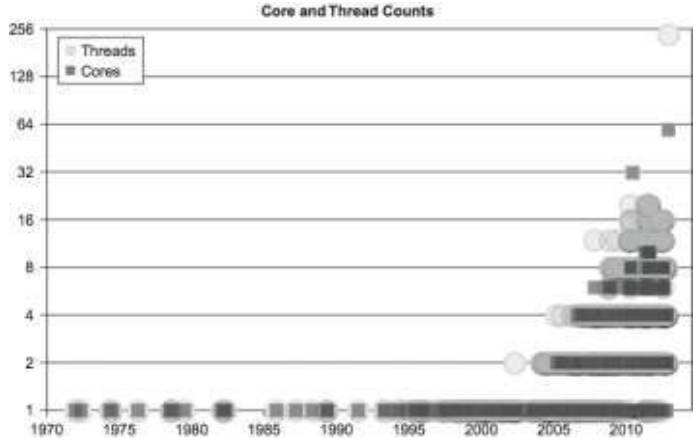


“Писать паралельные программы и писать корректные паралельные программы - не одно и то же” :) (c) **Paul E. McKenney, CppCon 2015**

Параллелизм – норма жизни



- Мультиядерные и многоядерные процессоры
- SIMD, SIMT, SMT
- Степень аппаратного ||| продолжает расти
- Требуется явный ||| на уровне софта



- Здесь и далее:
  ||| означает параллелизм

“The Free Lunch Is Over”*

1. **The Power Wall**: Clock frequency cannot be increased without exceeding air cooling.
2. **The Memory Wall**: Access to data is a limiting factor.
3. **The ILP Wall**: All the existing instruction-level parallelism (ILP) is already being used.

Conclusion: Explicit parallel mechanisms and explicit parallel programming are required for performance scaling.

“Concurrency is the next major revolution in how we write software” *

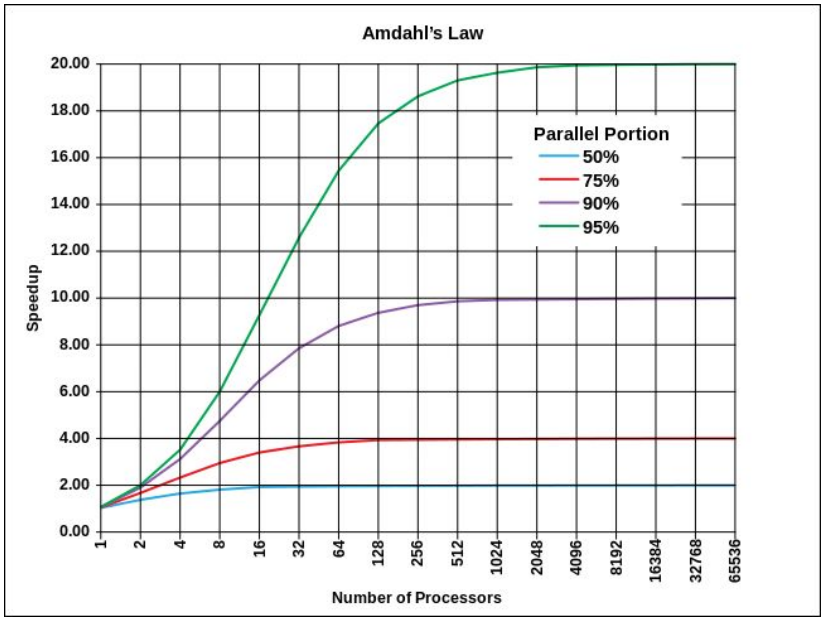
* © Herb Sutter, 2005

Speedup and Efficiency

- T_1 = time to run with 1 worker
- T_P = time to run with P workers
- T_1 / T_P = speedup
  - The relative reduction in time to complete the same task
  - Ideal case is linear in P
    i.e. 4 workers gives a best-case speedup of 4.
  - In real cases, speedup often significantly less
  - In rare cases, such as search, can be superlinear
- $T_1/(PT_P)$ = efficiency
  - 1 is perfect efficiency
  - Like linear speedup, perfect efficiency is hard to achieve
  - Note that this is not the same as “utilization”

Закон Амдала

«В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента»



$$S_p \leq \frac{1}{f + (1-f)/p} \leq S^* = \frac{1}{f}.$$

S_p - ускорение

p - число “процессоров”

f - последовательная часть

Закон Густавсона — Барсиса

"it may be most realistic to assume that *run time, not problem size*, is constant."

- John Gustafson. "Reevaluating Amdahl's Law" (*Communications of the ACM,* 31(5), 1988**)**

$$S_p = g + (1 - g)p = p + (1 - p)g$$

S_p - ускорение

p - число “процессоров”

g - последовательная часть

*Общая работа = g + **Np***

Если *g* - постоянно то Увеличивая “размер задачи” мы улучшаем маштабируемость!

C++ до 2011: ||| на свой страх и риск

- Стандарт языка C++ (1998)
  - Строго последовательная модель, ||| не рассматривается вовсе
- Разнообразные библиотеки потоков /*threads*/
  - POSIX threads, WinAPI threads, Boost, …
  - Чаще всего, кросс-платформенные обёртки над средствами ОС
- OpenMP
  - Языковое расширение на основе прагм, развиваемое консорциумом производителей
  - Ориентировано на HPC (Fortran,C); слабо интегрировано с C++
- «Молодая шпана»
  - TBB (Intel), PPL (Microsoft), GCD (Apple), CUDA (NVidia), Cilk++ (CilkArts => Intel), академические разработки, …
  - Потоки, если и используются, скрыты от программиста

C++11 и |||: необходимые основы

- Усовершенствованная модель памяти
- Многопоточная модель исполнения программы
- «Гонки данных» /*data races*/ объявлены UB
- Переменные, локальные для потока: thread_local
- Поддержка в библиотеке:
  - Потоки исполнения: std::thread
  - Атомарные переменные: std::atomic<>
  - Синхронизация: std::mutex и др., std::condition_variable
  - Асинхронное исполнение: std::future, std::async

Базовые блоки для многопоточных программ

Как это всё использовать?

Разработать:

- Управление потоками исполнения
- Распределение работы между потоками
- Синхронизацию при использовании разделяемых данных

Ключевые моменты:

- Накладные расходы
- Балансировку нагрузки
- Масштабируемость
- Компонуемость составных частей программы
- Переносимость

Общепринятое мнение:
Параллельные программы – это сложно

C++ - Questions about multithreading

1. Do I need to query or even consider the number of cores on a machine or when the threads are running, they are automatically sent to free cores?

2. Can anyone show me a

Assume we have an array or vector of length 256(can be more or less) and the number of pthreads to generate to be 4(can be more or less).

I need to figure out how to assign each pthread to a process a section

I have a large `for` loop, in which I want each item to be passed to a function on a thread. I have a thread pool of a certain size, and I want to reuse the threads. What is the most efficient way to do this?

Basically, I want all the threads to wait for the READY signal before continuing. num_thread is set to 0, and READY is false before threads are created. Once in a while, deadlock occurs. Can anyone help

When should I allocate a new thread to the task?

I have one task to compute 1(one array, and the second on

In this simplified code, all the threads may try to write the exact same value to the same memomry location in `vec`. Is this a data race likely to trigger undefined behavior, or is it safe since the values are never read before all the threads are joined again?

Многопоточность – это сложно

“Кто виноват ?”

- Слишком низкий уровень абстракции
- Отсутствие необходимых знаний и опыта
- Некоторые концепции **действительно** сложны!

“Что делать ?”

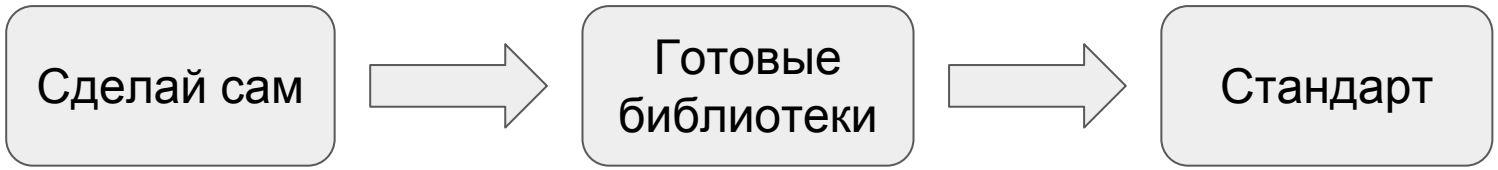
- Нанять эксперта в разработке многопоточных программ
- Стать таким экспертом

- **Использовать другие подходы к параллелизму**

Будьте экспертом в своей области!

Deja vu

- Мы это всё уже проходили, и не раз
  - В 1990-х: «оконный» интерфейс для DOS, класс string и т.д.
  - До появления STL: списки, очереди, ... и использующие их алгоритмы
- Не надо «изобретать велосипед»!
  - Используйте C++-библиотеки «параллельных шаблонов» /*parallel patterns*/
  - Есть выбор: от Intel, Microsoft, NVidia, AMD, Qualcomm, ...
  - Многие с открытым исходным кодом

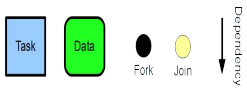


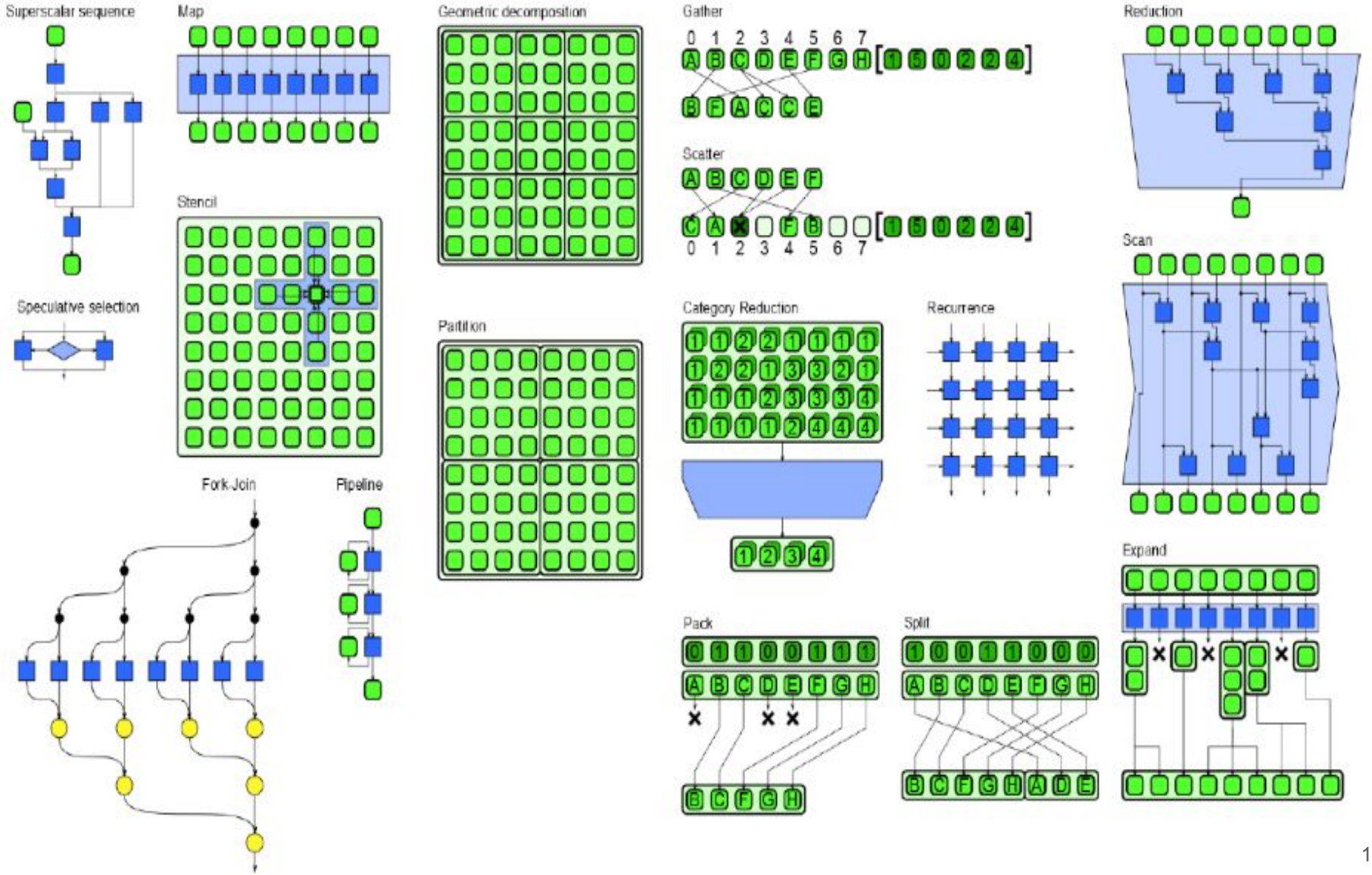
Параллелизм без потоков – это как?

- Определите ||| на алгоритмическом уровне
  - Разбейте алгоритм на отдельные вычислительные блоки
  - Определите зависимости между блоками
- Найдите подходящий параллельный шаблон
- Примените этот шаблон с помощью выбранной библиотеки
- Если нужного шаблона не нашлось, но есть API для использования «задач», попробуйте применить его
- Оставьте библиотеке сложную и рутинную работу
- Profit! :)

Параллельное программирование
может быть доступным !

Параллельные шаблоны





Intel® Threading Building Blocks

- Библиотека C++
  - Портируемое решение
  - Основана на C++ шаблонах /*templates*/
- ||| в виде задач
  - Что исполнять параллельно - а не как
  - Балансировка методом перехвата работы
- Параллельные алгоритмы
  - Типовые шаблоны параллелизма
  - Эффективная реализация
- Конкурентные контейнеры
  - Контейнеры в стиле STL
  - Не требуют внешних блокировок
- Примитивы синхронизации
  - Мьютексы с разными свойствами
  - Атомарные операции
- Масштабируемый менеджер памяти
  - • Спроектирован для параллельных программ

Шаблоны /*Patterns*/

Structured Programming with Patterns

- Patterns are “best practices” for solving specific problems.
- Patterns can be used to organize your code, leading to algorithms that are more scalable and maintainable.
- A pattern supports a particular “algorithmic structure” with an efficient implementation.
- Good parallel programming models support a set of useful parallel patterns with low-overhead implementations.

Structured Serial Patterns

The following patterns are the basis of **“structured programming”** for serial computation:

- Sequence
- Selection
- Iteration
- Nesting
- Functions
- Recursion

- Random read
- Random write
- Stack allocation
- Heap allocation
- Objects
- Closures

Using these patterns, “goto” can (mostly) be eliminated and the maintainability of software improved

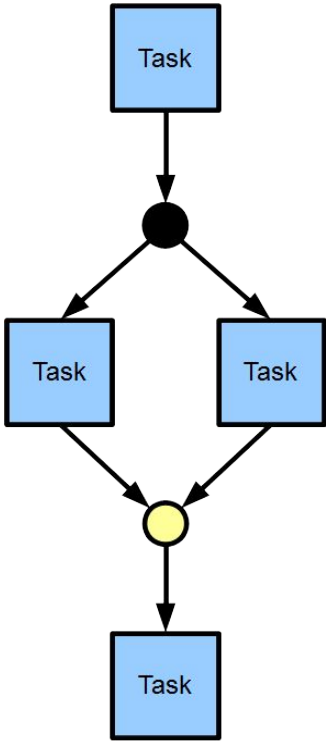
Structured Parallel Patterns

The following additional parallel patterns can be used for **“structured parallel programming”**:

- Superscalar sequence
- Speculative selection
- Map
- Recurrence
- Scan
- Reduce
- Pack/expand
- Fork/join
- Pipeline
- Partition
- Segmentation
- Stencil
- Search/match
- Gather
- Merge scatter
- Priority scatter
- *Permutation scatter
- !Atomic scatter

Using these patterns, threads and vector intrinsics can (mostly) be eliminated and the maintainability of software improved

Шаблон: Fork-Join



Fork-Join

- • Fork-join запускает исполнение нескольких задач одновременно и затем дожидается завершения каждой из них
- Удобен в применении для функциональной и рекурсивной декомпозиции
- Используется как базовый блок для построения других шаблонов

Примеры: Сортировка слиянием, быстрая сортировка (Хоара), другие алгоритмы «разделяй-и-властвуй»

Fork-Join в Intel (R) TBB

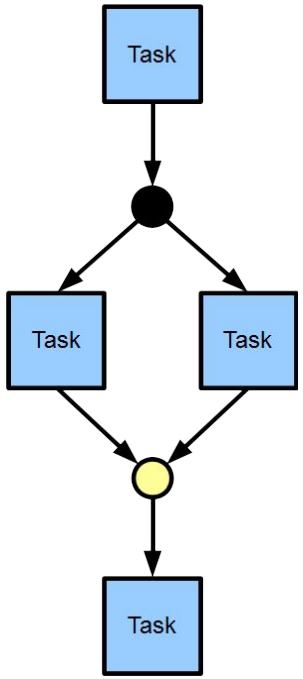
Для небольшого предопределённого кол-ва задач

```
parallel_invoke( a, b, c );
```

Когда кол-во задач велико или заранее неизвестно

```
task_group g;
g.run( a );
g.run( b );

...
g.run_and_wait( c );
```



Fork-Join

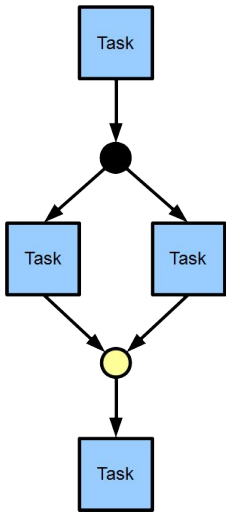
Fork-Join. Пример: быстрая сортировка

```
template<typename I>
void fork_join_qsort(I begin, I end)
{
    typedef typename std::iterator_traits<I>::value_type T;
    if (begin != end) {
        const I pivot = end - 1;
        const I middle = std::partition(begin, pivot,
              std::bind2nd(std::less<T>(), *pivot));
        std::swap(*pivot, *middle);
        tbb::parallel_invoke(
              fork_join_qsort(begin, middle),
              fork_join_qsort(middle + 1, end)
        );
    }
}
```

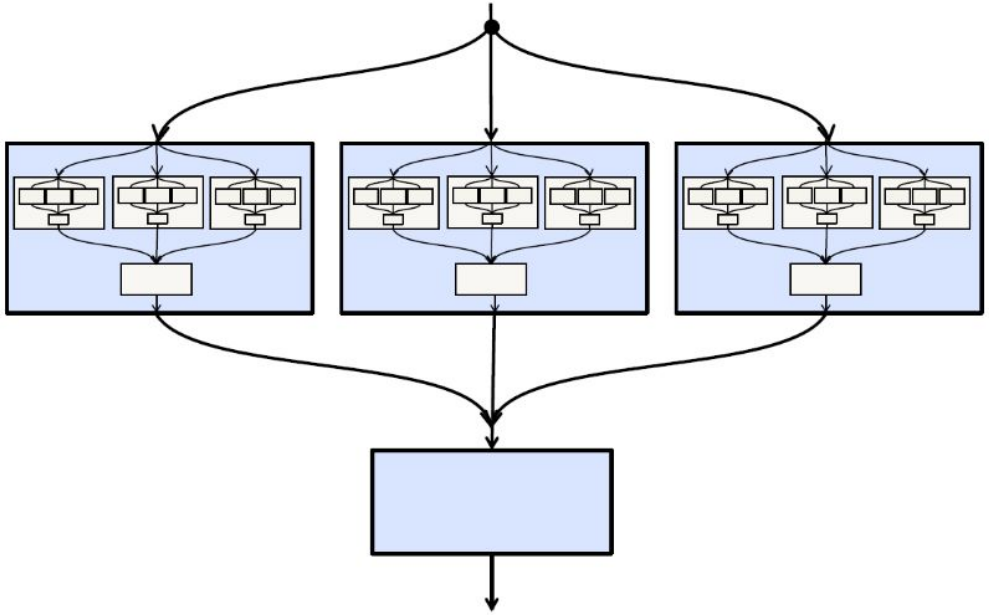
Recursive Patterns

- • Recursion is an important “universal” serial pattern
  - Recursion leads to functional programming
  - Iteration leads to procedural programming
- Structural recursion: nesting of components
- Dynamic recursion: nesting of behavior

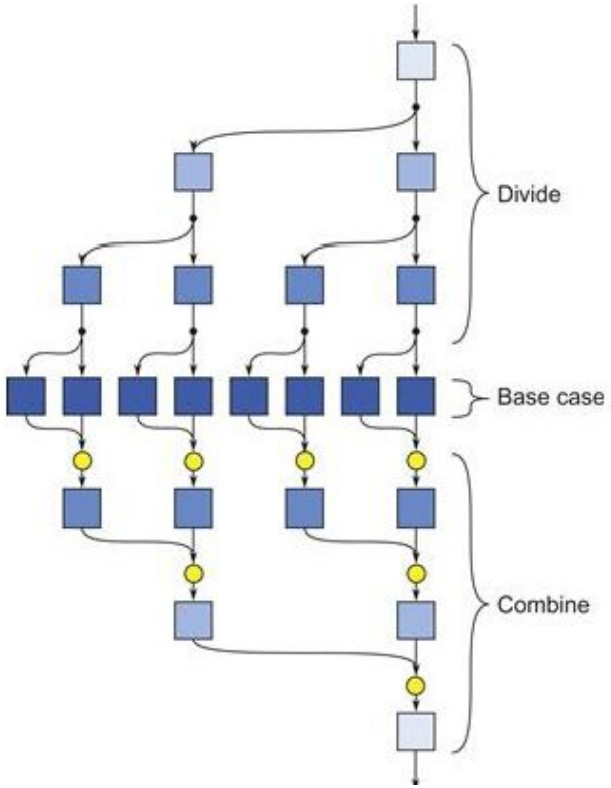
Рекурсивный (вложенный) параллелизм



Fork-Join



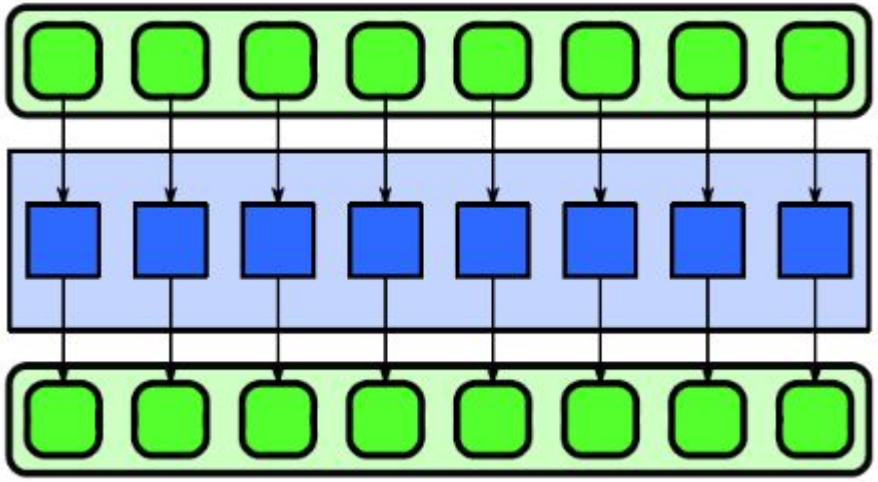
Эффективная рекурсия с fork-join



- Легко «вкладывается»
- Накладные расходы делятся между потоками
- Именно так устроены tbb:: parallel_for, tbb:: parallel_reduce

Рекурсивный fork-join обеспечивает высокую степень параллелизма

Шаблон: Map



- Применение элементной функции к множеству значений параллельно
- Это может быть некий набор данных или абстрактный индекс
  A = map(f)(B);

- В серийной программе это частный случай итерирования – независимые операции

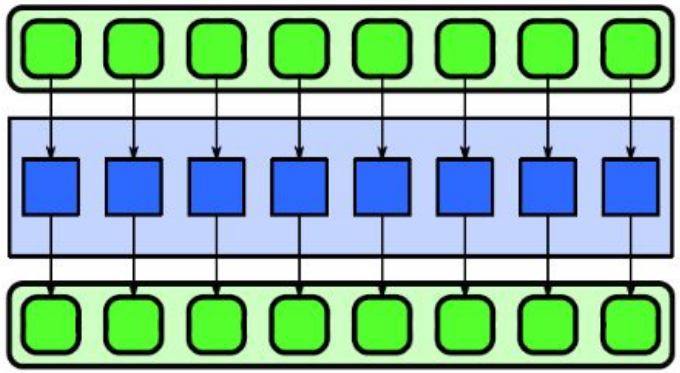
Примеры: цветовая коррекция изображений; преобразование координат; трассировка лучей; методы Монте-Карло

Шаблон: Map. Использование

```
parallel_for( 0, n, [&]( int i ) {
        a[i] = f(b[i]);
});
```

```
parallel_for(
        blocked_range<int>(0,n),
        [&](blocked_range<int> r ) {
                for( int i=r.begin(); i!=r.end(); ++i )
                        a[i] = f(b[i]);

        });
```



tbb::parallel_for

- Предоставляется в нескольких вариантах

Применить *functor(i)* ко всем *i* из множества *[lower,upper)*

parallel_for(*lower, upper, functor*);

Применить *functor(i)*, изменяя *i* с заданным шагом *stride*

parallel_for(*lower, upper, stride, functor*);

Применить *functor(subrange)* для набора *subrange* из *range*

parallel_for(*range, functor*);

Map; parallel_for; Пример: SAXPY

Scaled Vector Addition (SAXPY):

$y = a \times x + y$ $\quad$ x, y – векторы, a - скаляр

Часто встречается в линейной алгебре (Название из библиотеки BLAS, Basic Linear Algebra Subprograms, http://www.netlib.org/blas/)

```
void saxpy_serial(
        float a,                                // скалярный множитель
        const float x[],                        // первый исходный вектор
        float y[],                              // второй исходный и результирующий
вектор
        int n                                   // количество элементов
)
{
        for (int i = 0; i < n; i++)
                y[i] = a * x[i] + y[i];
}
```

Map; parallel_for; Пример: SAXPY

```
void saxpy( float a, float x[], float y[], size_t n )
{
	tbb::parallel_for( size_t(0), n, [&]( size_t i ) {
		y[i] += a*x[i];
	});
}
```

```
void saxpy( float a, float x[], float y[], size_t n )
{
	size_t gs = std::max( n/1000, 1 );
	tbb::parallel_for( tbb::blocked_range<size_t>(0,n,gs),
	[&]( tbb::blocked_range<size_t> r ) {
		for( size_t i=r.begin(); i!=r.end(); ++i )
			a[i] = f(b[i]);
	}, tbb::simple_partitioner() );
}
```

Управление распределением работы

Рекурсивное деление на максимально возможную глубину

parallel_for(*range, functor,* **simple_partitioner**());

Глубина деления подбирается динамически

parallel_for(*range, functor,* **auto_partitioner**());

Деление запоминается и по возможности воспроизводится

affinity_partitioner affp**;**
parallel_for(*range, functor,* affp);

Map; parallel_for; Пример ||| в 2D

```
// serial
for( int i=0; i<m; ++i )
        for( int j=0; j<n; ++j )
                a[i][j] = f(b[i][j]);
```

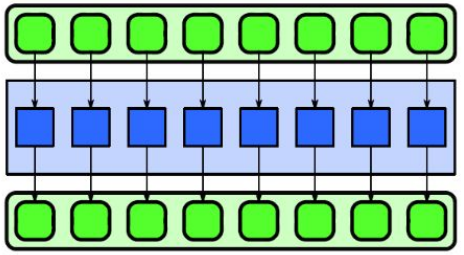
Декомпозиция «плиткой» /*tiling*/ в 2D может приводить к лучшей локальности данных, чем вложенные ||| циклы в 1D.

```
tbb::parallel_for(
        tbb::blocked_range2d<int>(0,m,0,n),
        [&](tbb::blocked_range2d<int> r ) {
                for( int j=r.rows().begin(); j!=r.cols().end(); ++j )
                        for( int i=r.rows().begin(); i!=r.rows().end(); ++i )
                                a[i][j] = f(b[i][j]);
        }
);
```

Если parallel_for не подходит

Применить *functor(*iter)* ко всем элементам контейнера

parallel_for_each(*first, last, functor*);



- Параллельная версия std::for_each
- Работает со стандартными контейнерами

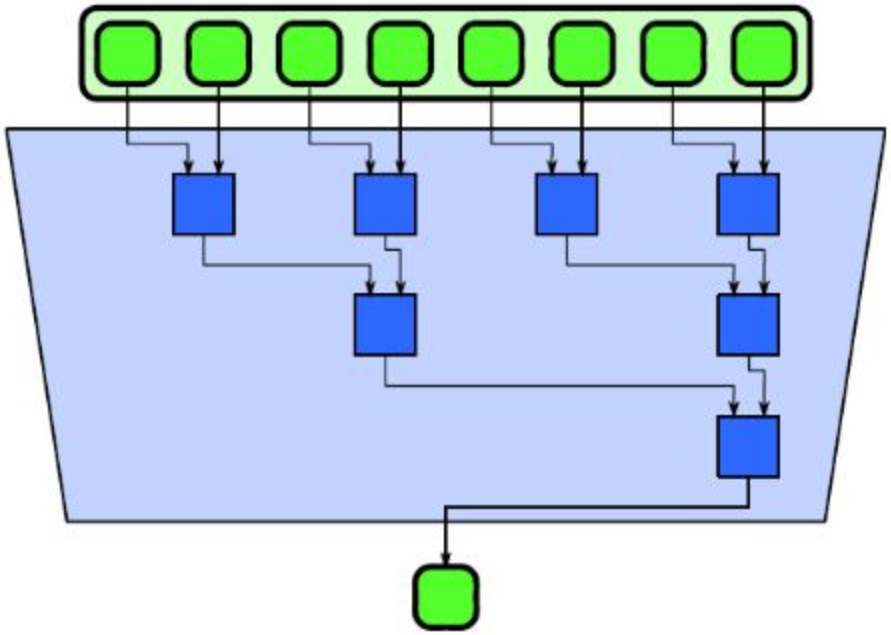
То же с возможностью добавить работу «на лету»

parallel_do(*first, last, functor*)**;**

Добавление данных для обработки:

```
[]( work_item, parallel_do_feeder& feeder ){
      <обработка полученного work_item>
      if( <в процессе создан new_work_item> )
            feeder.add( new_work_item );
};
```

Шаблон: Reduce /*свёртка*/



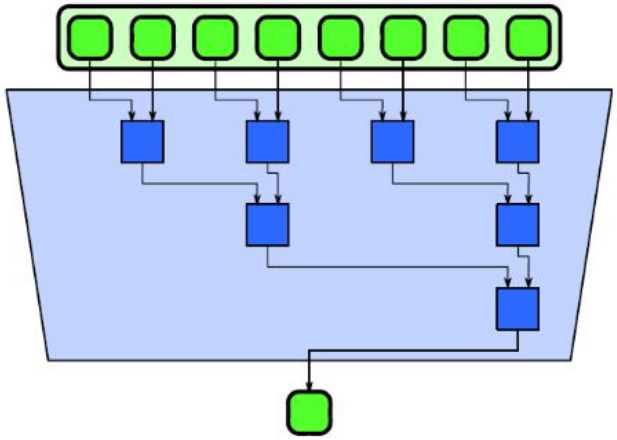
- *Reduce* объединяет, при помощи ассоциативной операции, все элементы набора в один элемен
- Это может быть некий набор данных или абстрактный индекс
  b = reduce(f)(B);

Примеры: вычисление агрегатных функций; операции с матрицами; численное интегрирование

- Например, *reduce* можно использовать, чтобы найти сумму элементов или максимальный эл-т

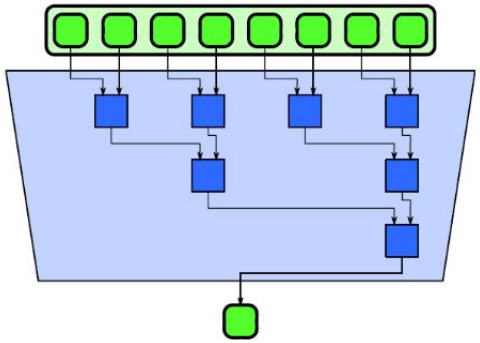
Шаблон: Reduce. Использование

```
enumerable_thread_specific<float> sum;
parallel_for( 0, n, [&]( int i ) {
        sum.local() += a[i];
});
... = sum.combine(std::plus<float>());
```



```
sum = parallel_reduce(
        blocked_range<int>(0,n),
        0.f,
        [&](blocked_range<int> r, float s) -> float
        {
                for( int i=r.begin(); i!=r.end(); ++i )
                        s += a[i];
                return s;
        },
        std::plus<float>()
);
```

Пример с enumerable_thread_specific



- Подходит, если:
  - Операция коммутативна
  - Дорого вычислять свёртку (напр. большой размер операндов)

Контейнер для thread-local представлений

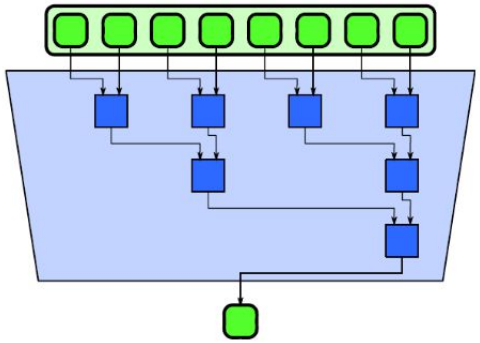
Обращение к локальной копии

```
enumerable_thread_specific<BigMatrix> sum;
…
parallel_for( 0, n, [&]( int i ) {
        sum.local() += a[i];
});
... = sum.combine(std::plus<BigMatrix>());
```

Применяет указанную операцию для свёртки локальных копий

Пример с parallel_reduce

- Подходит, если:
  - Операция некоммутативна
  - Использование диапазонов /*range*/ улучшает производительность



Нейтральный элемент

Свёртка поддиапазона

Свёртка частичных реультатов

Начальное значение для свёртки

```
sum = parallel_reduce(
      blocked_range<int>(0,n),
      0.f,
      [&](blocked_range<int> r, float s) -> float
      {
            for( int i=r.begin(); i!=r.end(); ++i )
                  s += a[i];
            return s;
      },
      std::plus<float>()
);
```

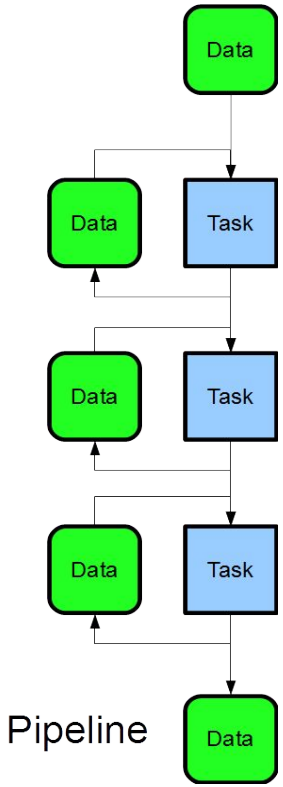
Reduce. Пример: поиск наименьшего элемента

```
// Find index of smallest element in a[0...n-1]
int ParallelMinIndex ( const float a[], int n ) {
    struct MyMin {float value; int idx;};
    const MyMin identity = {FLT_MAX,-1};
    MyMin result = tbb::parallel_reduce(
        tbb::blocked_range<int>(0,n),
        identity,
        [&] (tbb::blocked_range<int> r, MyMin current) -> MyMin {
            for( int i=r.begin(); i<r.end(); ++i )
                if(a[i]<current.value ) {
                    current.value = a[i];
                    current.idx = i;
                }
            return current;
        },
        [] (const MyMin a, const MyMin b) {
            return a.value<b.value? a : b;
        });
    return result.idx;
}
```

Комментарии к parallel_reduce

- Можно указывать необязательный аргумент *partitioner*
  - аналогично **parallel_for**
- Для неассоциативных операций рекомендуется **parallel_deterministic_reduce**
  - Воспроизводимый результат для арифметики с плавающей точкой
  - Но не соответствует результату в серийном коде
  - Рекомендуется явно указывать гранулярность
  - Не позволяет задать partitioner

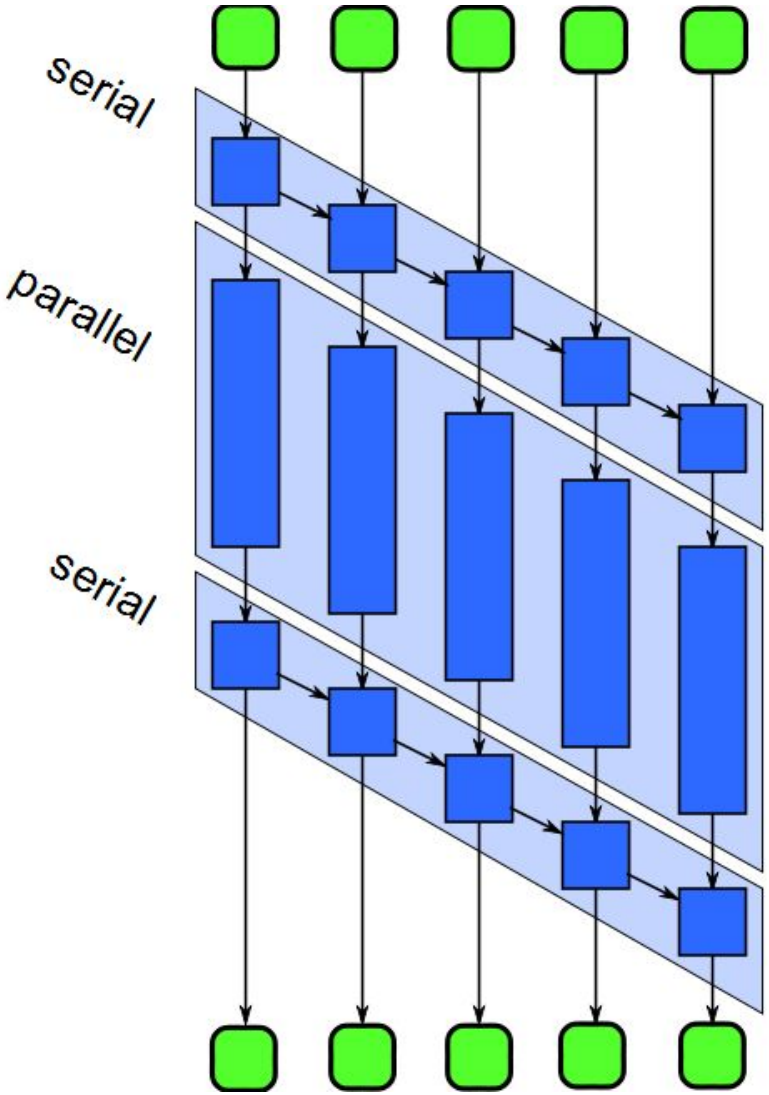
Шаблон: Pipeline /*конвейер*/



- *Конвейер* – цепочка из стадий обработки потока данных
- Некоторые стадии могут иметь состояние
- Можно обрабатывать данные по мере поступления: “online

Примеры: сжатие/распаковка данных, обработка сигналов, фильтрация изображений

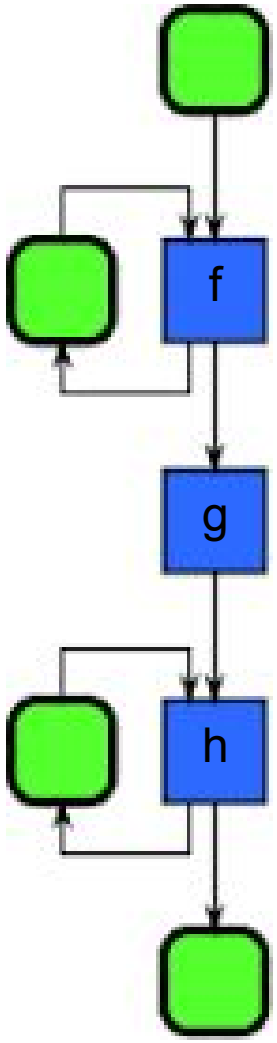
||| в конвейере



- Разные данные на разных стадиях
- Разные данные в одной стадии, если там нет состояния
  - Данные на выходе могут быть переупорядочены
- Может понадобиться буферизация между стадиями

Шаблон: Pipeline. Использование

```
parallel_pipeline (
        ntoken,
        make_filter<void,T>(
                filter::serial_in_order,
                [&]( flow_control & fc ) -> T{
                        T item = f();
                        if( !item ) fc.stop();
                        return item;
                }
        ) &
        make_filter<T,U>(
                filter::parallel,
                g
        ) &
        make_filter<U,void>(
                filter:: serial_in_order,
                h
        )
);
```



Стадии конвейера

Параллельная стадия – функциональное преобразование

Преобразование X в Y

Серийная стадия может поддерживать состояние

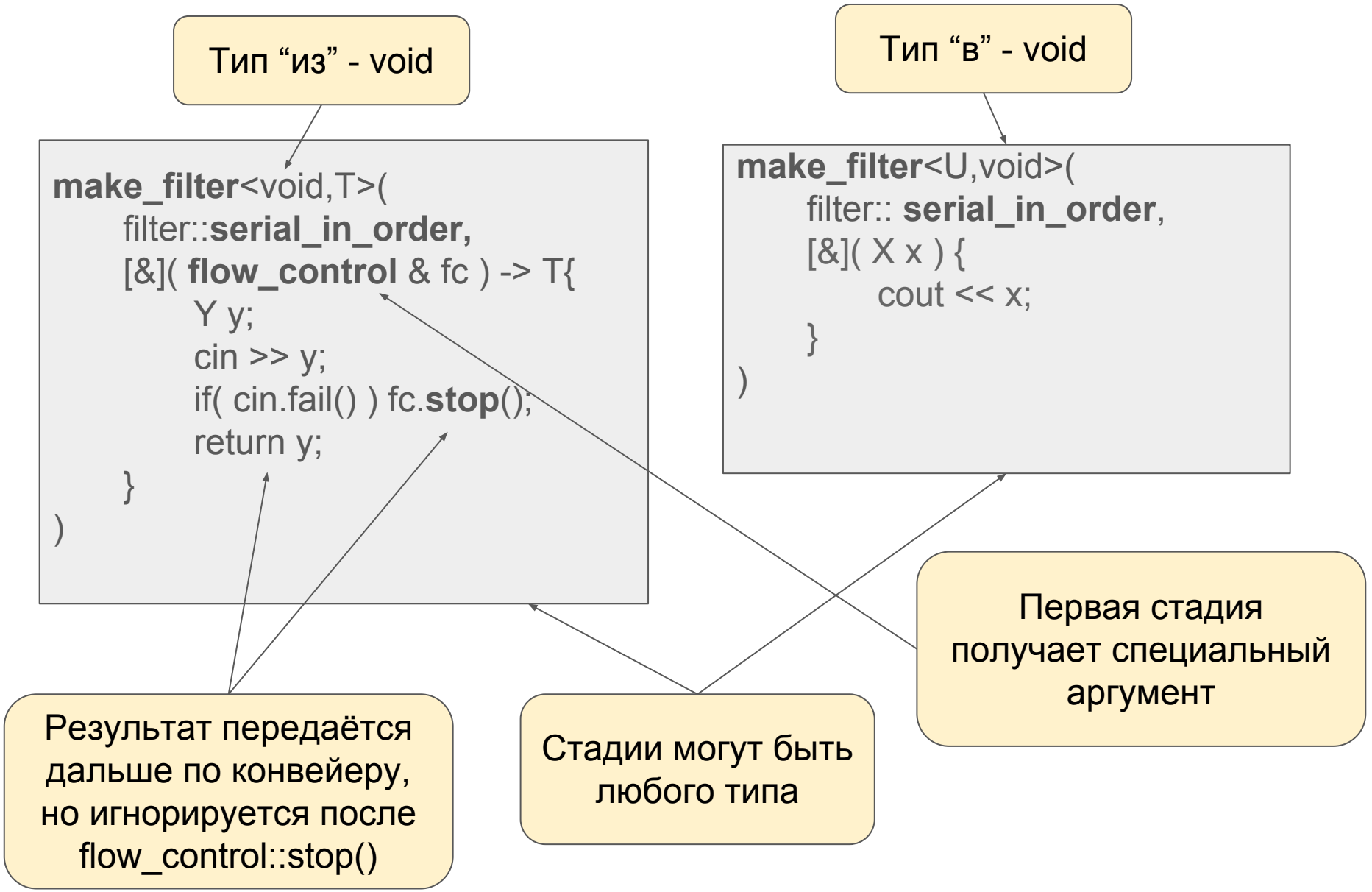
```
make_filter<X,Y>(
        filter::parallel,
        []( X x ) -> Y {
                Y y = foo(x);
                return y;
        }
)
```

```
make_filter<X,Y>(
        filter::serial_in_order,
        [&]( X x ) -> Y {
                extern int count;
                ++count;
                Y y = bar(x);
                return y;
        }
)
```

Отсутствие «гонок» – ответственность программиста

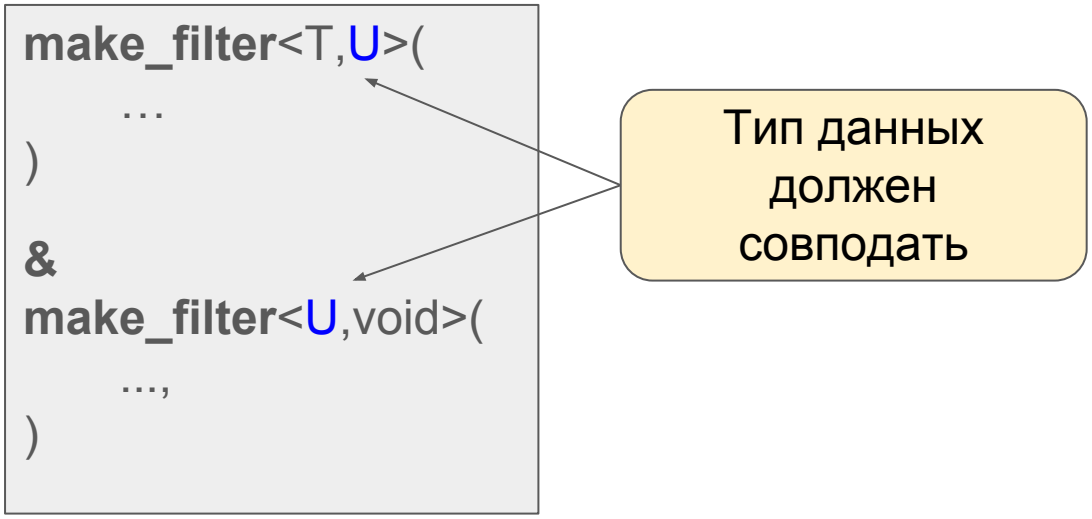
Данные поступают в порядке, заданном на предыдущей упорядоченной стадии

Стадии конвейера: вход и выход



Построение конвейера

- Стадии соединяются при помощи **operator&**



Алгебра типов

make_filter<*T,U*>(*mode,functor*) -> filter_t<*T,U*>

filter_t<*T,U*> & filter_t<*U,V*> -> filter_t<T,V>

Запуск конвейера

parallel_pipeline (size_t *ntoken*, const filter_t<void, void> & *filter*);

Ограничение на кол-во данных в обработке

Цепочка стадий void -> void.

Эффективное использование кэша

- Один поток проводит данные через множество этапов
- Предпочтение обработке имеющихся элементов

Масштаби руемость

- Функциональная декомпозиция не масштабируется
- Параллельные стадии улучшают ситуацию
- Производительность ограничена серийными стадиями

Снижение сложности с ||| шаблонами

Не нужно беспокоиться	• Об управлении потоками исполнения • О распределении работы • О компонуемости параллельных частей • О переносимости на другое «железо»
Масштаби руемость	• Функциональная декомпозиция не масштабируется • Параллельные стадии улучшают ситуацию • Производительность ограничена серийными стадиями

Параллельные программы – это доступно !

Заключение

- Эффективно использовать современные процессоры невозможно без параллельного программирования
- Поддержка параллелизма в C++ пока лишь на базовом уровне: потоки, синхронизация и т.п.
- Тема ||| активно развивается в C++ Standard Committee
- Для эффективной и продуктивной разработки применяйте готовые решения
  - Библиотеки параллельных шаблонов
  - Эффективные языковые расширения (если приемлемо)
  - Специализированные библиотеки с поддержкой |||

Успехов в параллельном программировании!

Источники информации

- T.G.Mattson, B.A.Sanders, B.L.Massingill:
  Patterns for Parallel Programming,
  Addison-Wesley, 2005, ISBN 978-0-321-22811-6
- M.McCool, A.D.Robinson, J.Reinders:
  Structured Parallel Programming,
  Morgan Kaufmann, ISBN 978-0-12-415993-8 www.
  parallelbook.com
- Intel® Threading Building Blocks, www.threadingbuildingblocks.
  org