МНОГОПОТОЧНОСТЬ И ПАРАЛЛЕЛИЗМ В C++

Курсы Intel Delta-3

Алексей Куканов, Intel Corporation

Нижний Новгород, 2015

Часть 2. Параллельные шаблоны

(на примере Intel® Threading Building Blocks)

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»!

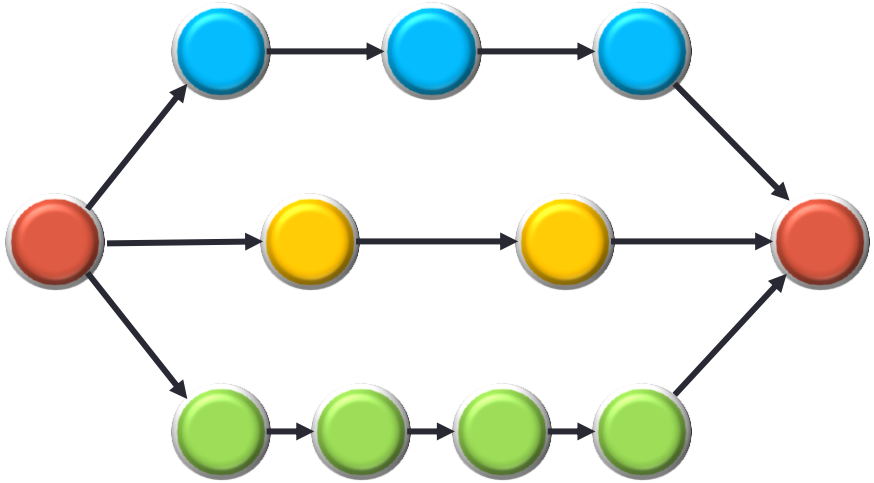
- Используйте C++-библиотеки «параллельных шаблонов» /*parallel patterns*/
- Есть выбор: от Intel, Microsoft, NVidia, AMD, Qualcomm, …
- Многие с открытым исходным кодом

Сделай сам → Готовые библиотеки → Стандарт

Параллелизм: простое объяснение

В геометрии параллельные прямые не пересекаются

В программировании параллельные задачи не взаимодействуют

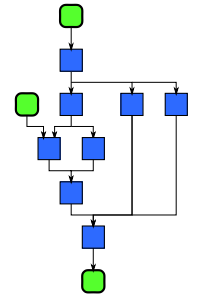


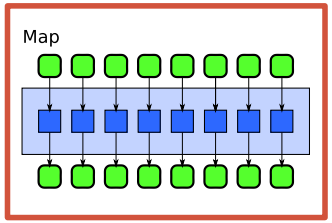
Параллелизм без потоков – это как?

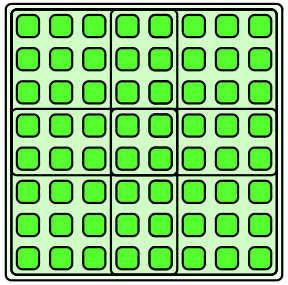
- Определите ||| на алгоритмическом уровне
  - Разбейте алгоритм на отдельные вычислительные блоки
  - Определите зависимости между блоками
- Найдите подходящий параллельный шаблон
- Примените этот шаблон с помощью выбранной библиотеки
- Если нужного шаблона не нашлось, но есть API для использования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

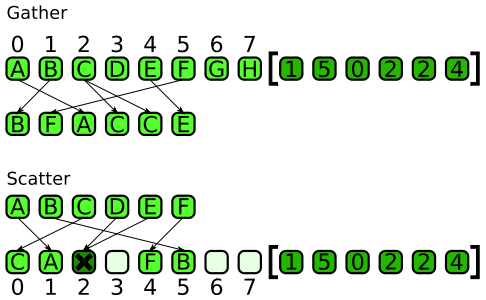
Параллельное программирование может быть доступным

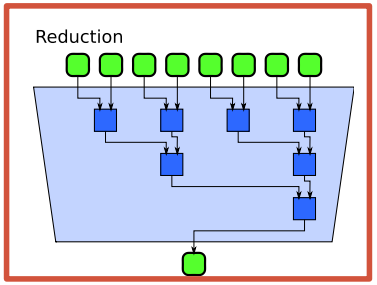
Параллельные шаблоны

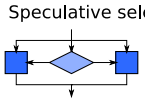


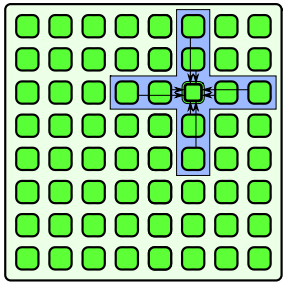


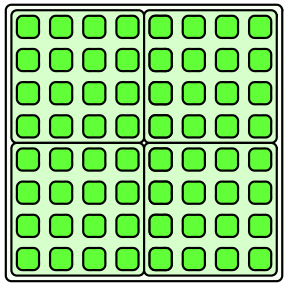


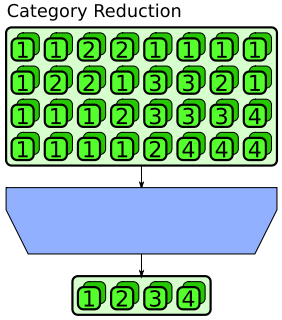


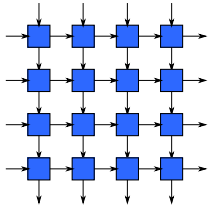


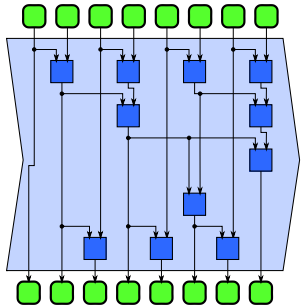


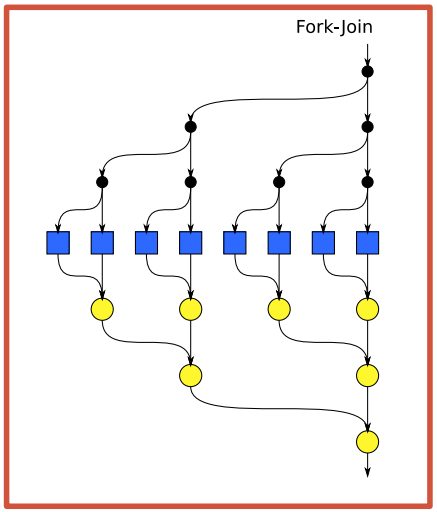


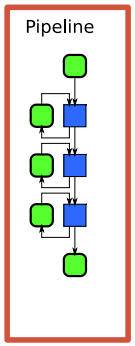


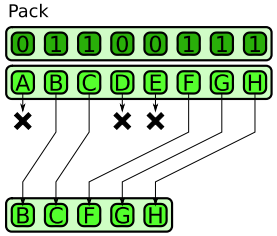


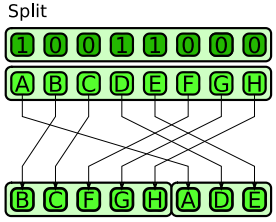


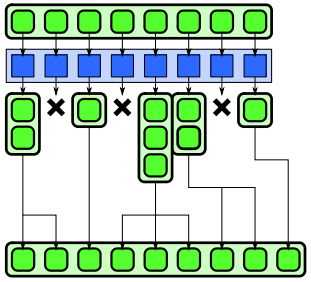








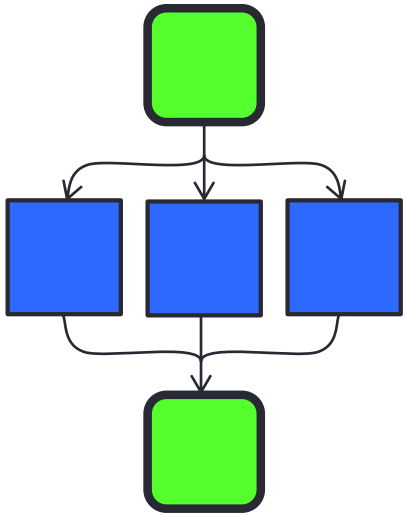




Intel® Threading Building Blocks

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

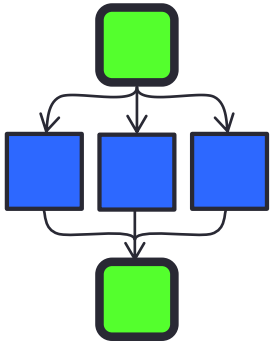
Шаблон: Fork-Join



- *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® TBB



Для небольшого предопределённого кол-ва задач

```
parallel_invoke( functor1, functor2, ...);
```

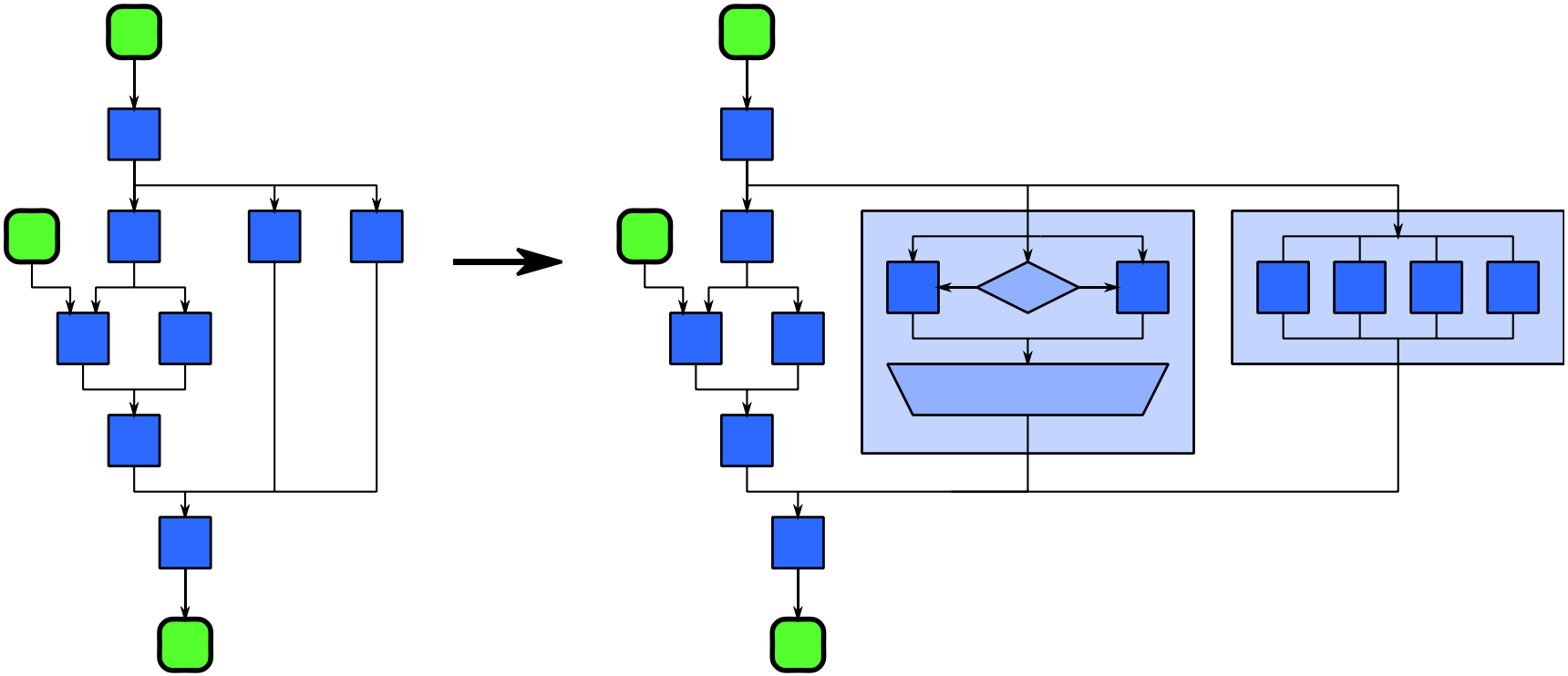
Когда кол-во задач велико или заранее неизвестно

```
task_group g;
...
g.run( functor1 );
...
g.run( functor2 );
...
g.wait(); // also run_and_wait()
```

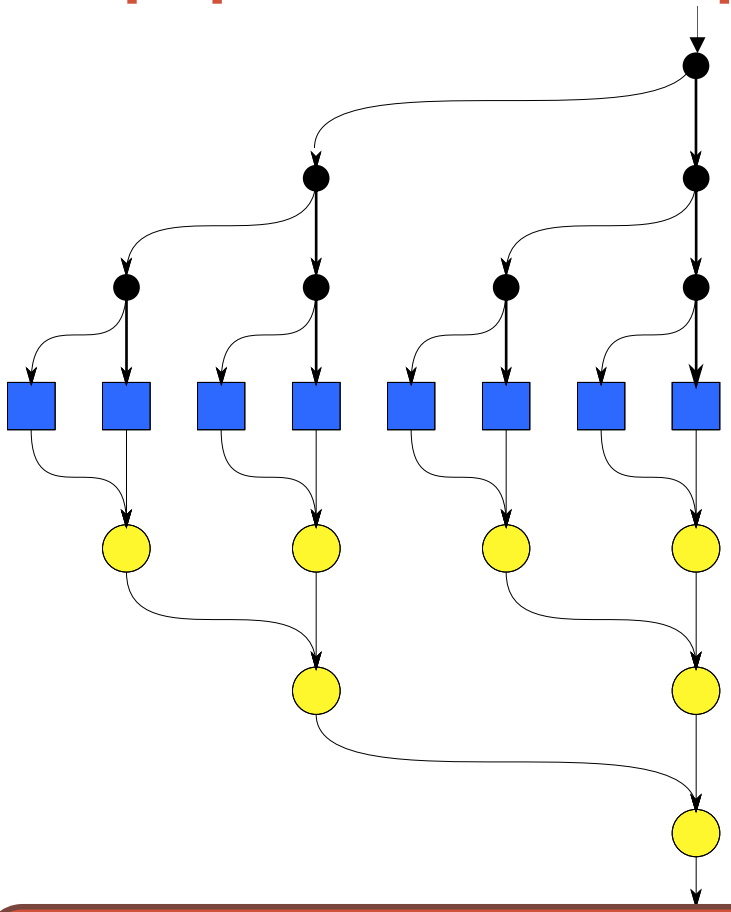
Пример: быстрая сортировка

```
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)
    );
  }
}
```

Рекурсивный (вложенный) параллелизм



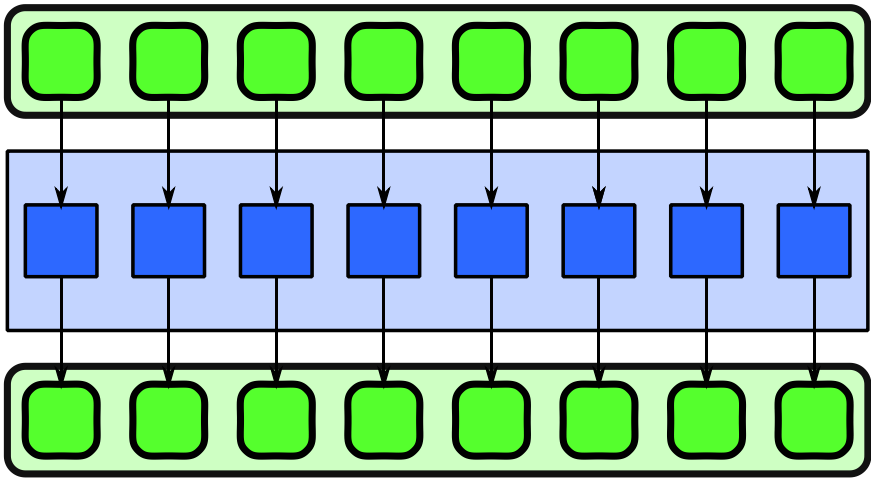
Эффективная рекурсия с fork-join



- 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



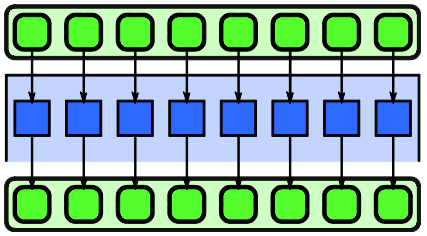
- *Map* применяет указанную функцию к каждому элементу из заданного набора
- Это может быть некий набор данных или абстрактный индекс

```
A = map(f)(B);
```

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

- В серийной программе это частный случай итерирования – независимые операции.

tbb::parallel_for



- Предоставляется в нескольких вариантах

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

```
parallel_for( lower, upper, functor );
```

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

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

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

```
parallel_for( range, functor );
```

Пример с parallel_for

```
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 )
            y[i] += a*x[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 );
```

Ещё пример: ||| в 2D

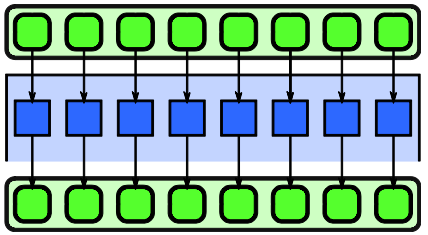
```
// serial
for( int i=0; i<m; ++i )
    for( int j=0; j<n; ++j )
        a[i][j] = f(b[i][j]);
```

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 i=r.rows().begin(); i!=r.rows().end(); ++i )
            for( int j=r.rows().begin(); j!=r.cols().end(); ++j )
                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

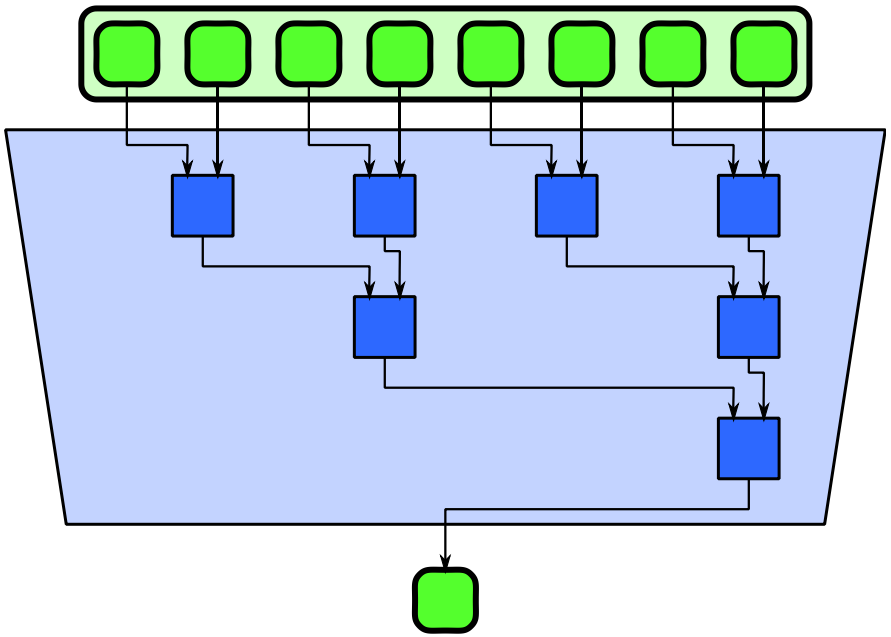
То же с возможностью добавить работу «на лету»

```
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*/



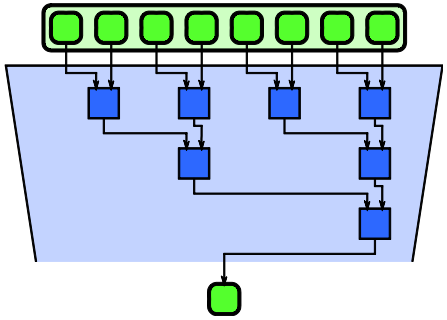
- *Reduce* объединяет, при помощи ассоциативной операции, все элементы набора в один элемент

```
b = reduce(f)(B);
```

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

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

Reduce в Intel® TBB



При помощи класса enumerable_thread_specific

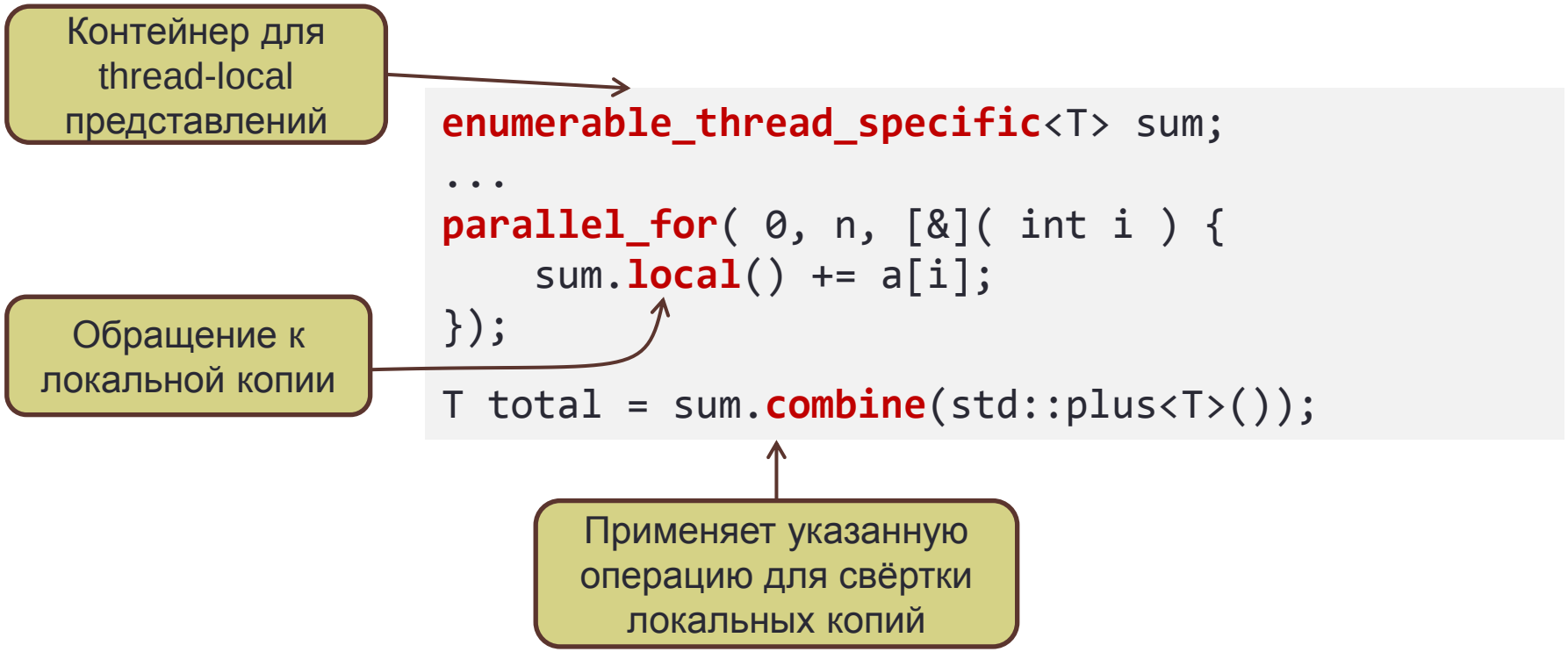
```
enumerable_thread_specific<T> sum;
parallel_for( 0, n, [&]( int i ) {
    sum.local() += a[i];
});
T total = s.combine(std::plus<T>());
```

При помощи функции parallel_reduce

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

Способ с enumerable_thread_specific

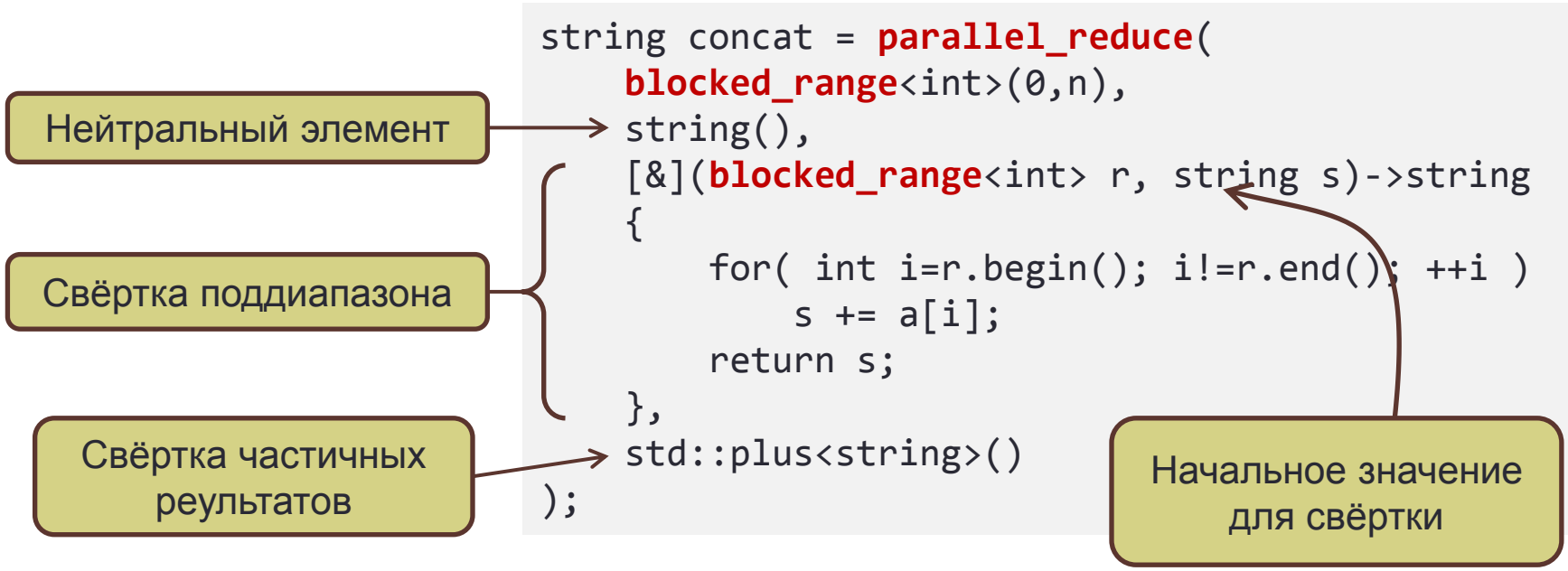
- Подходит, если:
  - Операция коммутативна
  - Дорого вычислять свёртку (напр. большой размер операндов)




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

Способ с parallel_reduce

- Подходит, если
  - Операция некоммутативна, но ассоциативна
  - Использование диапазона улучшает производительность




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

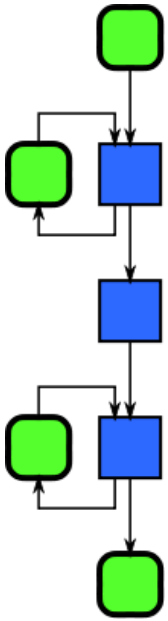
Пример: поиск наименьшего элемента

```
// 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*

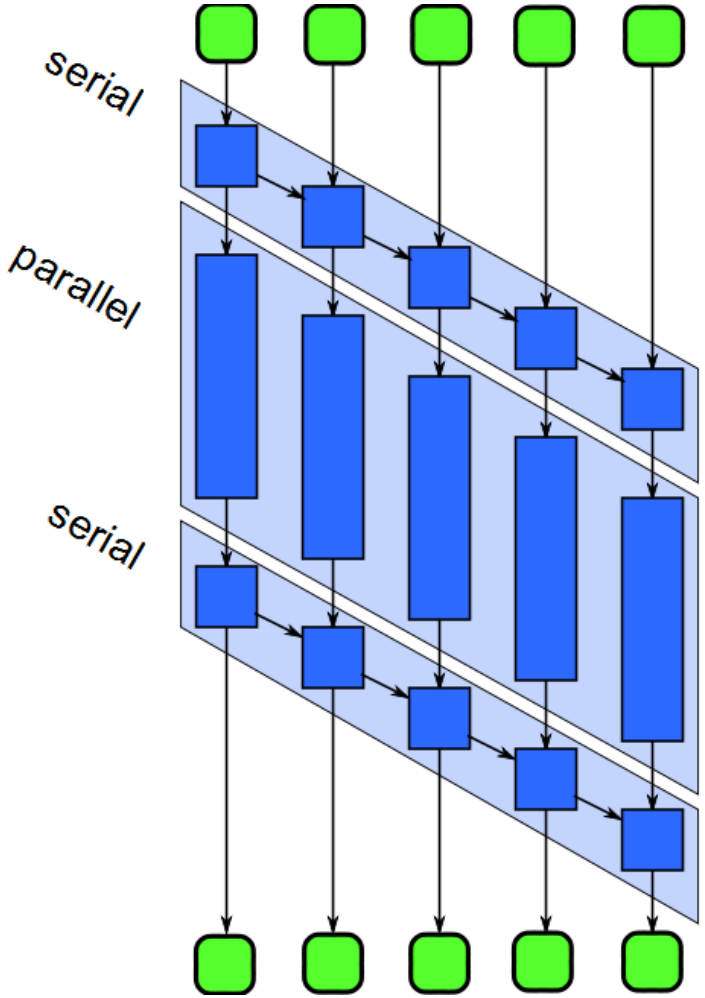
Шаблон: Pipeline /*конвейер*/



- *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

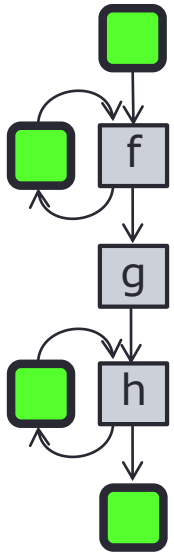
||| в конвейере



- 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

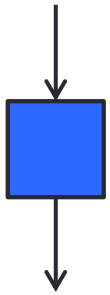
Pipeline в Intel® TBB

```
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
    )
);
```

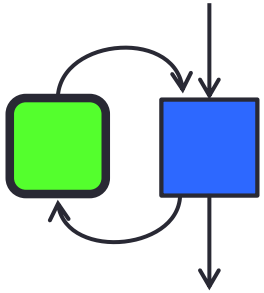


Стадии конвейера

Параллельная стадия – функциональное преобразование



Серийная стадия может поддерживать состояние



Преобразование X в Y

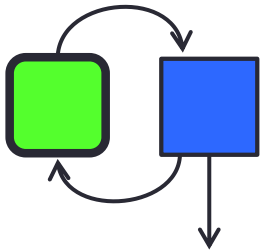
```
make_filter<X,Y>(
    filter::parallel,
    []( X x ) -> Y {
        Y y = foo(x);
        return y;
    }
)
```

Отсутствие «гонок» – ответственность программиста

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

Данные поступают в порядке, заданном на предыдущей упорядоченной стадии

Стадии конвейера: вход и выход



Тип “из” - **void**

```
make_filter<void,Y>(
    filter::serial_in_order,
    [&]( flow_control& fc ) -> Y {
        Y y;
        cin >> y;
        if( cin.fail() ) fc.stop();
        return y;
    }
)
```

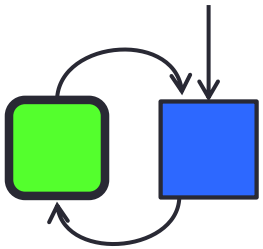
Первая стадия получает специальный аргумент

Стадии могут быть любого типа

Результат передаётся дальше по конвейеру, но игнорируется после **flow_control::stop()**

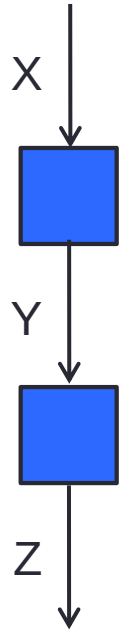
Тип “в” - **void**

```
make_filter<X,void>(
    filter::serial_in_order,
    [&]( X x ) {
        cout << x;
    }
)
```



Построение конвейера

- Стадии соединяются при помощи **operator&**




```
make_filter<X,Y>(
    ...
)
&
make_filter<Y,Z>(
    ...
)
```

Тип данных должен совпадать

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.

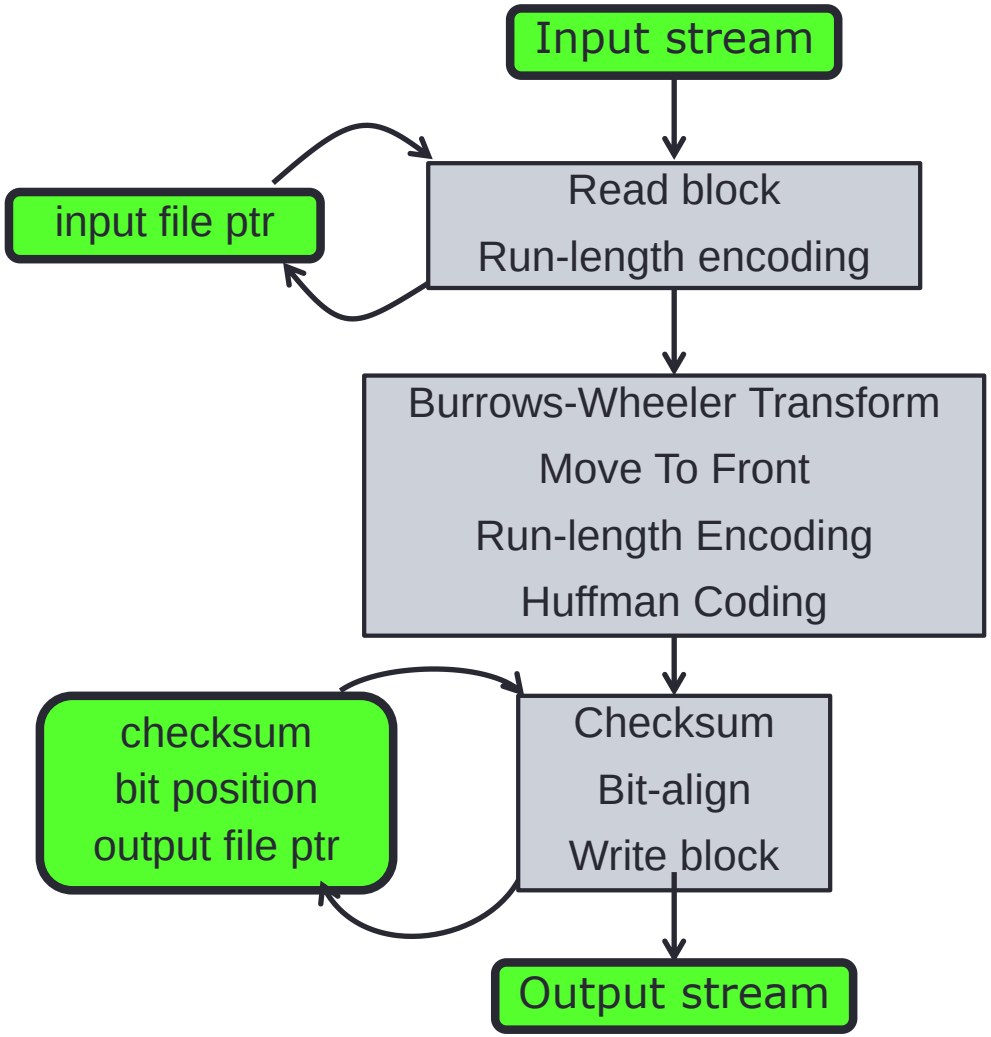
Эффективное использование кэша

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

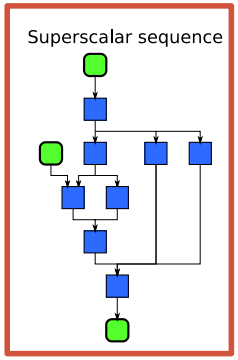
Масштаби-руемость

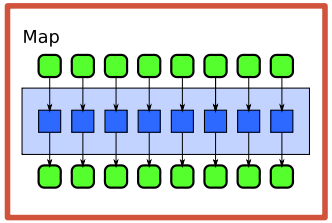
- Функциональная декомпозиция не масштабируется
- Параллельные стадии улучшают ситуацию
- Производительность ограничена серийными стадиями

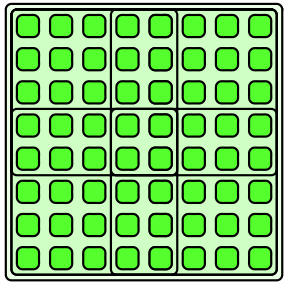
Bzip2: схема конвейера

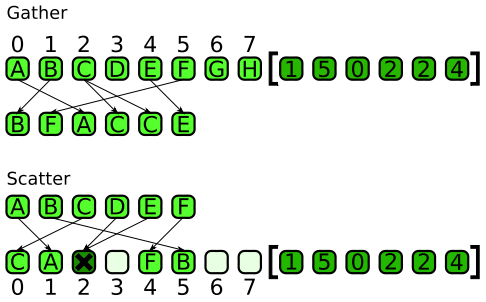


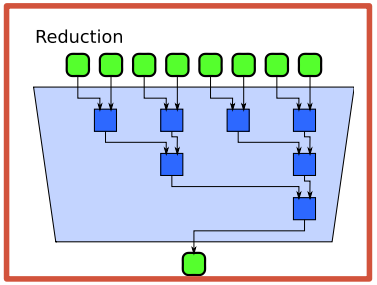
Параллельные шаблоны

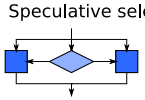


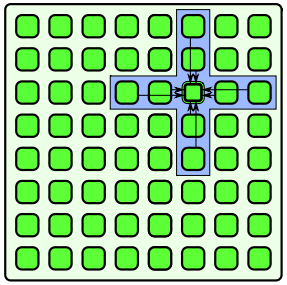


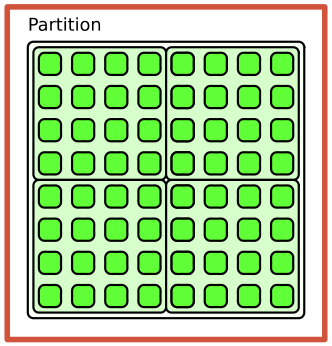


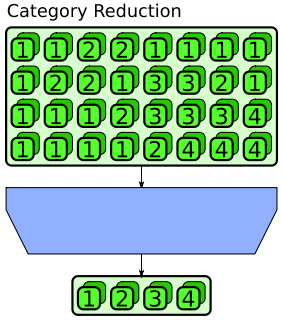


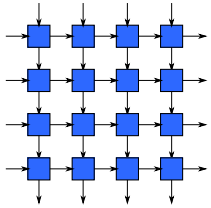


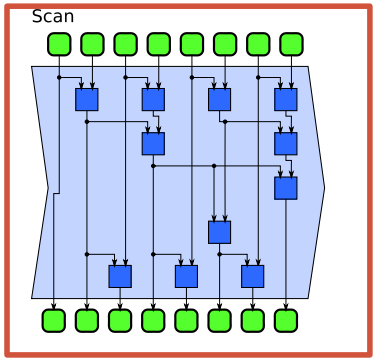


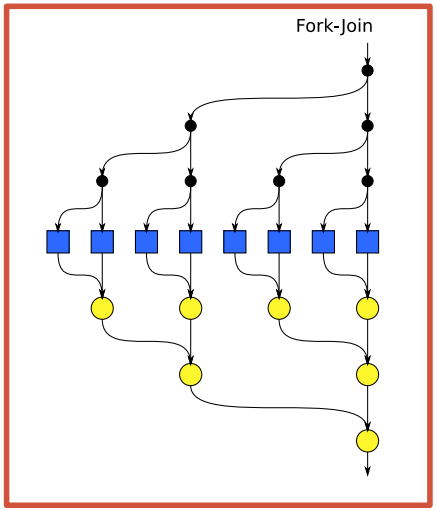


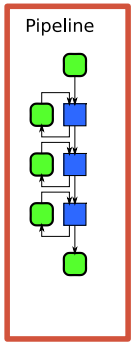


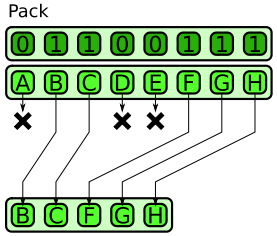


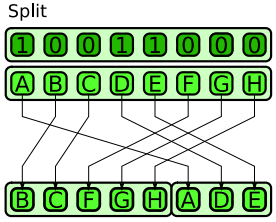


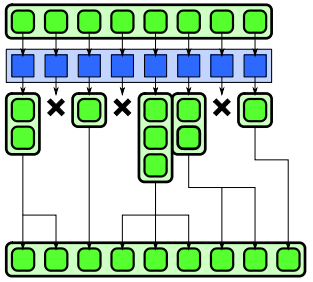












Снижение сложности с ||| шаблонами

Не нужно беспокоиться

- Об управлении потоками исполнения
- О распределении работы
- О компонуемости параллельных частей
- О переносимости на другое «железо»

Снижается необходимость

- В синхронизации
- В контроле накладных расходов
- В балансировке нагрузки

Параллельные программы – это доступно

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

Домашнее задание

1. Перепишите программы из предыдущего домашнего задания (подсчёт кол-ва слов в тексте, вычисление скалярного произведения) с использованием шаблонов алгоритмов в Intel® Threading Building Blocks

Продолжение следует...