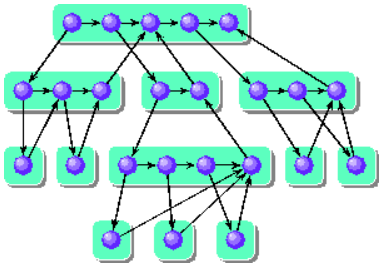
Лекция 9
Язык параллельного программирования Intel Cilk Plus

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Высокопроизводительные вычислительные системы»
Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)
Осенний семестр, 2015

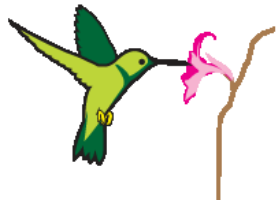
Язык программирования Cilk



- **Cilk** – расширение языка ANSI C для создания многопоточных программ
- **The Cilk Project:** разработка начата в 1994 г. в Лаборатории компьютерных наук Массачусетского института технологий (MIT, USA)
- Один из разработчиков и руководителей проекта – Charles E. Leiserson
- В 2006 г. C.E. Leiserson основал компанию **Cilk Arts** для продвижения на рынок языка Cilk
- Cilk-5.4.6 is the latest official MIT Cilk release (GPL)
- В 2007 г. выходит Cilk++ (Cilk с поддержкой C++)

http://supertech.csail.mit.edu/cilk/

Язык программирования Intel Cilk Plus



- В 2009 г. компания Intel приобрела Cilk Arts и выпустила **Intel Cilk Plus**
- **Intel Cilk Plus** – расширение языков C и C++
  - Параллелизм задач
  - Параллелизм по данным
- Intel поддерживает Cilk Plus в Development-ветви GCC 4.8 и GCC 4.9
  https://www.cilkplus.org/build-gcc-cilkplus
- Поддержка в Development-ветви LLVM/clang
  http://cilkplus.github.io/

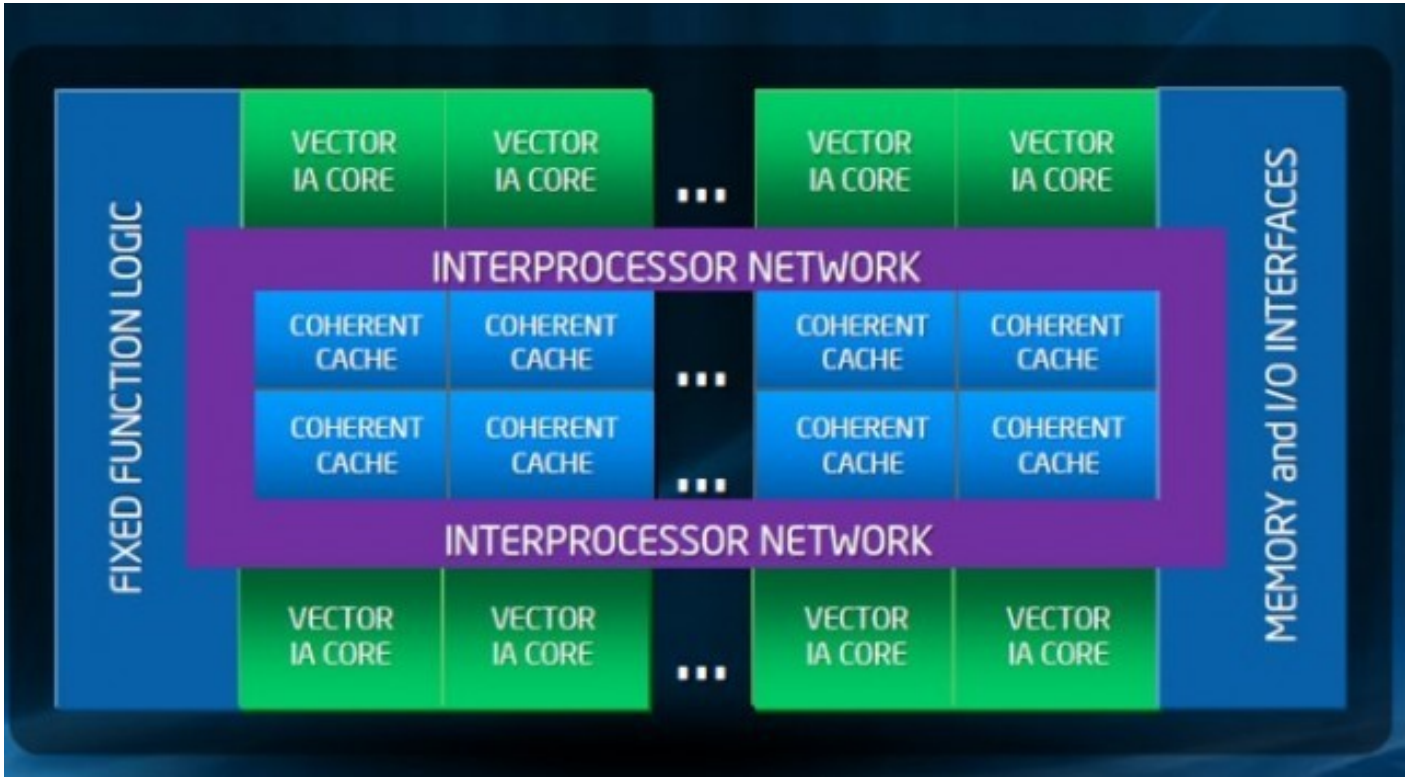
https://www.cilkplus.org

Intel Cilk Plus

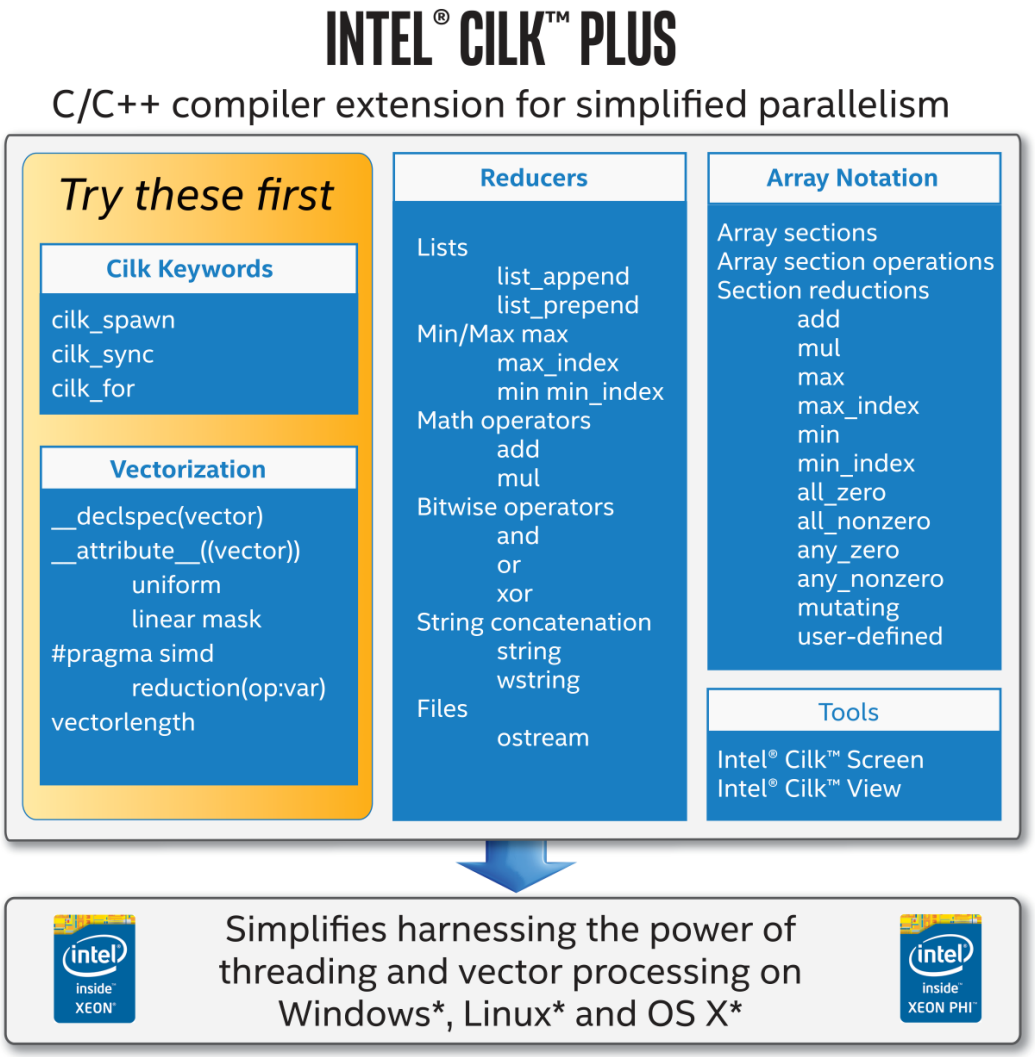
- **Компиляторы с поддержкой Cilk Plus**
  - **Intel C++ Compiler** (Intel Composer XE 2010)
  - **GCC** 5.0 (full support), 4.8 & 4.9 branches: http://gcc.gnu.org/svn/gcc/branches/cilkplus-4_8-branch
  - **Clang** Cilk Plus: http://cilkplus.github.io
- Intel Cilk Plus Runtime Library is Open Source (BSD-3) https://www.cilkplus.org/download
- **Open Specification**
  - Language Spec v1.2
  - ABI v1.1

Целевые архитектуры – SMP & NUMA

- Количество ядер и длины векторов увеличиваются (SSE4, AVX2, AVX-512)
- Intel Xeon Phi (Intel MIC):
  > 50 ядер x86, векторные регистры 512 бит



Intel Cilk Plus



https://software.intel.com/en-us/intel-cilk-plus

Intel Cilk Plus

- **Data parallelism**
  - ❑ Распараллеливание циклов: **_Cilk_for**
  - ❑ Конструкции для реализации редукции: Reducers
  - ❑ Векторная обработка массивов (Array notation) #pragma simd, elemental functions
- **Task parallelism**
  - ❑ Порождение задач (tasks): **_Cilk_spawn**
  - ❑ Синхронизация задач: **_Cilk_sync**
- Средствами Cilk Plus эффективно распараллеливаются:
  - алгоритмы типа “разделяй и властвуй” (divide & conquer): сортировка слиянием, быстрая сортировка и т.д.
  - операции над рекурсивными структурами данных (деревья, списки)

Язык Intel Cilk Plus

Keywords	
_Cilk_spawn	Сообщает компилятору, что функция может быть выполнена параллельно с вызывающей функцией
_Cilk_sync	Текущая функция не может продолжать выполнение дальше текущей точки параллельно с ее дочерними задачами (реализует синхронизацию, ожидание завершения дочерних задач)
_Cilk_for	Сообщает компилятору, что итерации цикла можно выполнять параллельно (каждой задаче достанется блок итераций размера *grainsize*)
Pragmas	
#pragma cilk grainsize = <expr>	Задает максимальное количество итераций цикла for для одной задачи
#pragma simd	Векторизует цикл средствами векторных инструкций процессора

Язык Intel Cilk Plus

Predefined Macro	
__cilk	Содержит номер поддерживаемой компилятором версии Intel Cilk Plus
Environment Variable	
CILK_NWORKERS	Задает количество рабочих потоков (worker threads), которые буду выполнять параллельные задачи
Elemental Function	
declspec(vector)	Конструкция для определения elemental functions
Compiler Options	
/Qcilk-serialize, -cilk-serialize	Компиляция программы в последовательном виде
/Qintel-extensions[-], -[no]intel-extensions	Включает/выключает расширения для Cilk Plus

Альтернативные ключевые слова

```
//
// file: <cilk/cilk.h>
//

#define cilk_spawn _Cilk_spawn
#define cilk_sync _Cilk_sync
#define cilk_for _Cilk_for
```

Intel Cilk Plus: Hello World!

```
#include <iostream>
#include <cilk/cilk.h>

void handler(int id, char *name)
{
    std::cout << "Hello World form " << id << ": "
              << name << std::endl;
}

int main()
{

    cilk_spawn handler(100, "Task X");
    cilk_spawn handler(120, "Task Y");
    cilk_spawn handler(130, "Task Z");
    cilk_sync;

    return 0;
}
```

Intel Cilk Plus: Hello World!

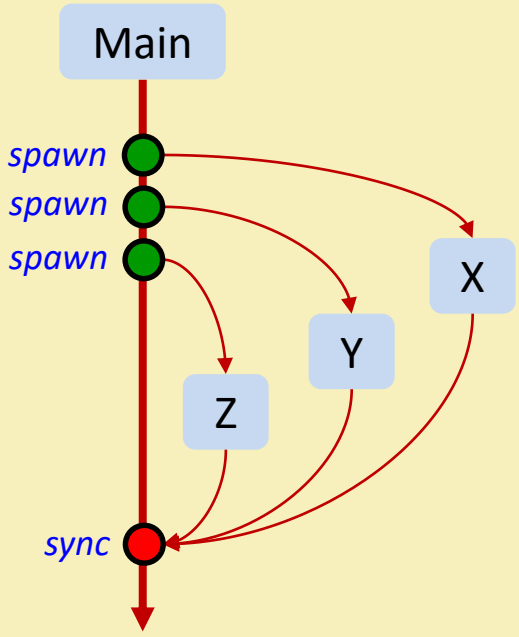
```
#include <iostream>
#include <cilk/cilk.h>

void handler(int id, char *name)
{
    std::cout << "Hello World form " << id << ": "
              << name << std::endl;
}

int main()
{

    cilk_spawn handler(100, "Task X");
    cilk_spawn handler(120, "Task Y");
    cilk_spawn handler(130, "Task Z");
    cilk_sync;

    return 0;
}
```

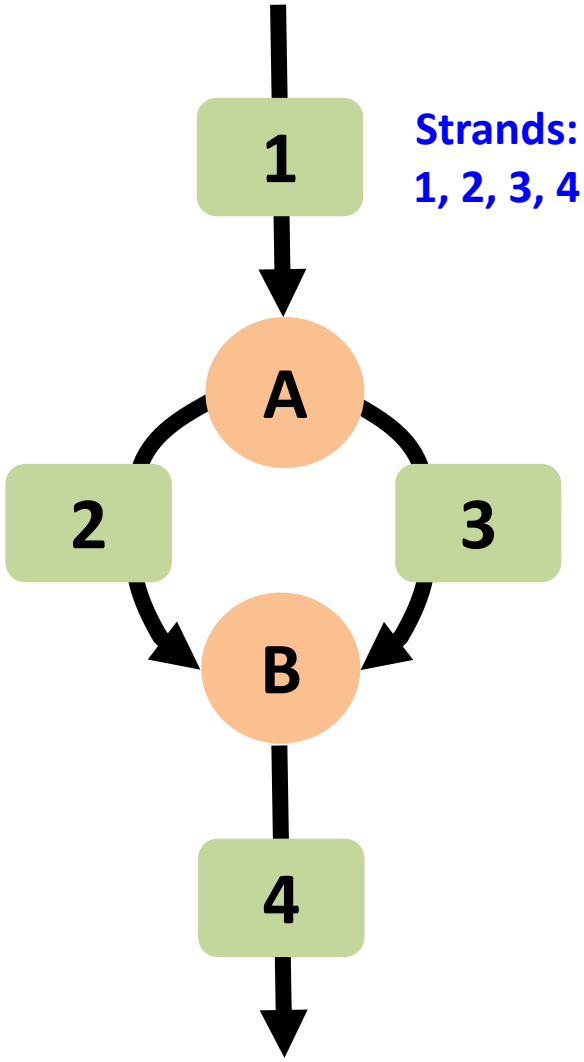


Intel Cilk Plus: Hello World!

```
$ icc –o prog ./prog.cpp
$ ./prog
Hello World form 100: Task X
Hello World form 120: Task Y
Hello World form 130: Task Z
```

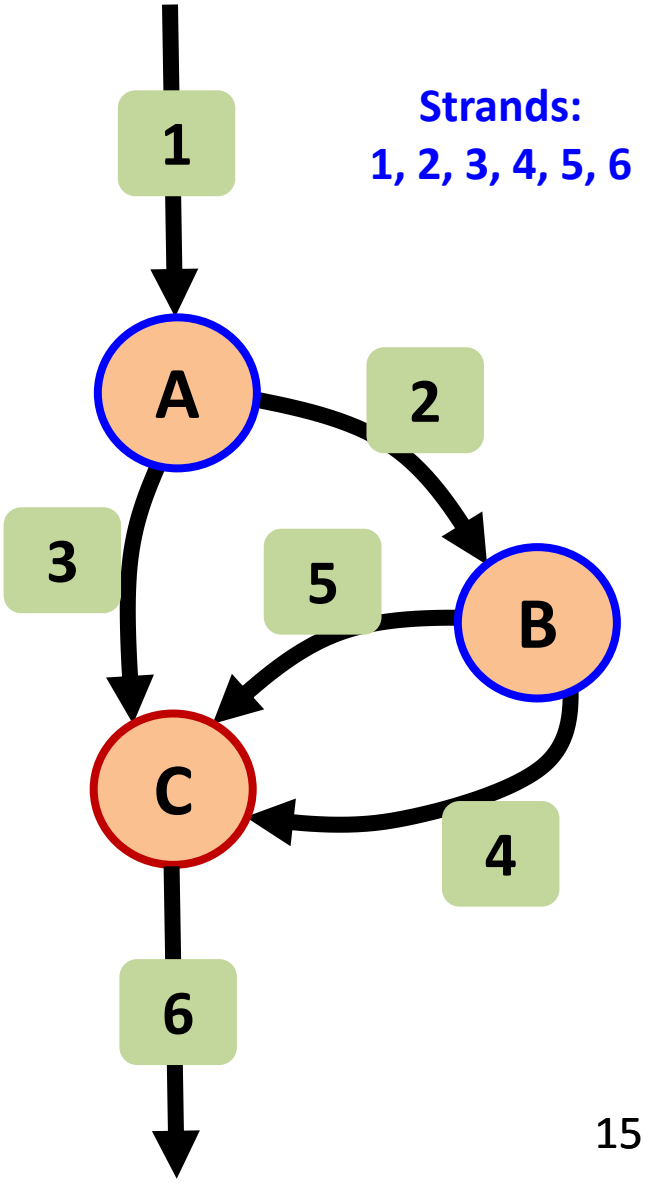
Основные термины Intel Cilk Plus

- **Strand** (нить) – последовательный участок кода (часть функции), без каких либо конструкций управления параллелизмом (spawn, sync)
- На рисунке приведено 4 strand
- Strands 2 и 3 могут выполняться параллельно – не обязательно будут выполнять параллельно!



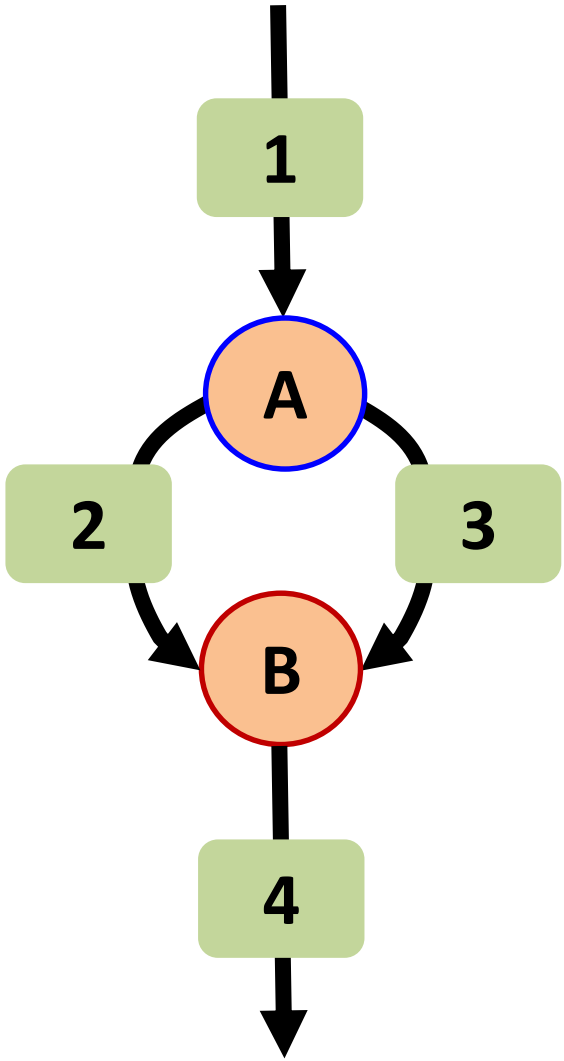
Представление программы в виде DAG

- Структура программа представляется направленным ациклическим графом **DAG** – Directed Acyclic Graph
- DAG не зависит от количества доступных процессоров (рабочих потоков, worker threads)
- Strands динамически распределяются планировщиком по пулу рабочих потоков (worker threads)
- На рисунке A, B – это операции spawn, C – это sync
- Strands 2 и 3 могут выполняться параллельно



Планировщик Intel Cilk Plus

```
{
    fun1();                 // strand 1
    _Cilk_spawn fun3();     // strand 3
    fun2();                 // strand 2
    _Cilk_sync;
    fun4();                 // strand 4
}
```



- **fun2** и **fun3** могут выполняться разными потоками
- Для обеспечения последовательной семантики (serial semantics), порожденная функция **fun3** всегда выполняется тем же потоком, который выполняет strand, вызвавший spawn
- В нашем случае **fun1** и **fun3** будут выполняться одним потоком
- Если есть свободные потоки, то strand2 (**fun2**) может быть выполнен им (захвачен, steal)

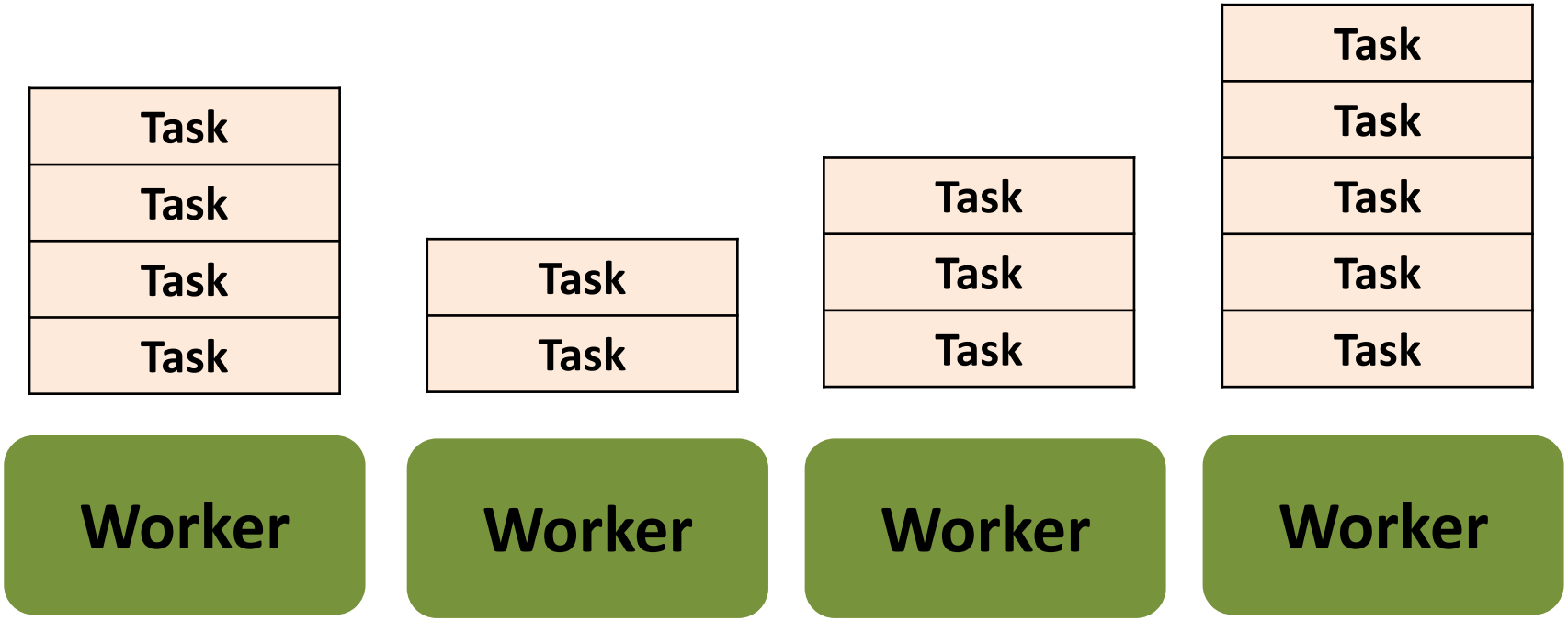
Intel Cilk Plus Execution Model

- Планировщик Cilk Plus динамически распределяет strands по потокам (workers)
- Функция, которая порождена нитью A,
  будет выполняться потоком (worker), который выполняет нить A
- Количество потоков в пуле можно настраивать:
  - `export CILK_NWORKERS=4`
  - `__cilkrts_set_param(), __cilkrts_get_nworkers(), __cilkrts_get_worker_number()`

```
if (0 != __cilkrts_set_param("nworkers", "4")) {
    printf("Failed to set worker count\n");
    return 1;
}
```

Планировщик Intel Cilk Plus

- Каждый поток имеет дек задач (deque, очередь с двумя концами)
- При порождении новой нити текущая помещается в очередь




```
export CILK_NWORKERS=4
```

Планировщик Intel Cilk Plus

- Планировщик использует механизм “work stealing” для динамического распределения strands по потокам (workers)
- Свободный поток (worker) может “захватить” (steal) у другого потока его strand
(выбирая с противоположного конца дека)

cilk_spawn

- **cilk_spawn** fun() – сообщает компилятору, что функция fun может быть выполнена параллельно с вызывающей функцией

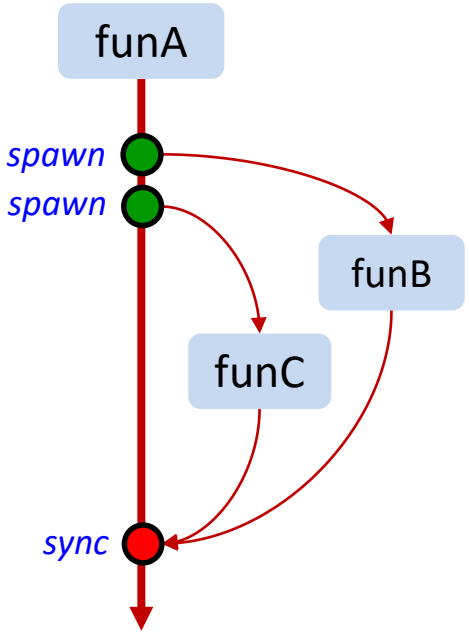
- В качестве функции fun могут быть указаны:
  - Обычная C-функция
  - Метод
  - Лямбда-функция C++11
  - Функтор (function object) – объект с перегруженным методом вызова функции operator()

- Параметры, передаваемые в порождаемую задачу вычисляются до вызова spawn

cilk_sync

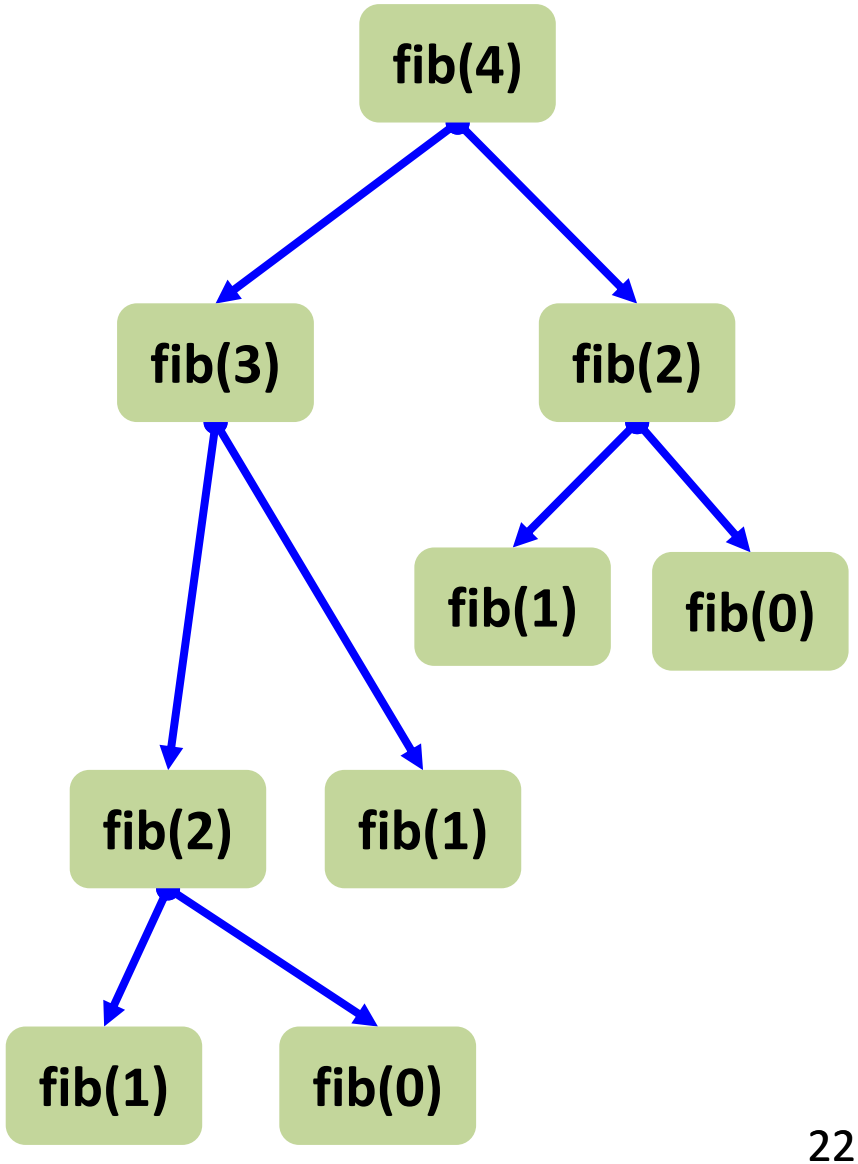
- **cilk_sync** – приостанавливает выполнение текущей нити (strand), пока все её дочерние нити не будут завершены

```
void funA()
{
    cilk_spawn funB();
    cilk_spawn funC();
    cilk_sync;
}
```



Числа Фибоначчи (Fibonacci)

```
int fib(int n)
{
    int x, y;
    if (n < 2)
        return n;
    x = fib(n - 1);
    y = fib(n - 2);
    return x + y;
}
```



$$F_n = F_{n-1} + F_{n-2}$$

0, 1, 1, 2, 3, 5, 8, …

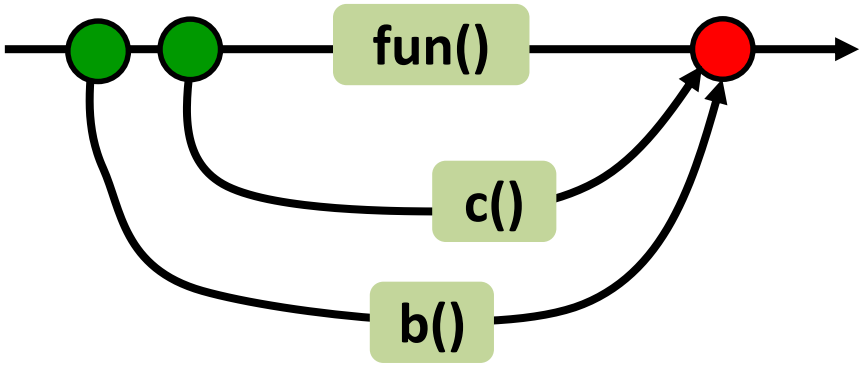
Числа Фибоначчи (Fibonacci)

```
int fib(int n)
{
    int x, y;
    if (n < 2)
        return n;
    x = cilk_spawn fib(n - 1);
    y = fib(n - 2);
    cilk_sync;
    return x + y;
}
```

cilk_spawn

Версия 1

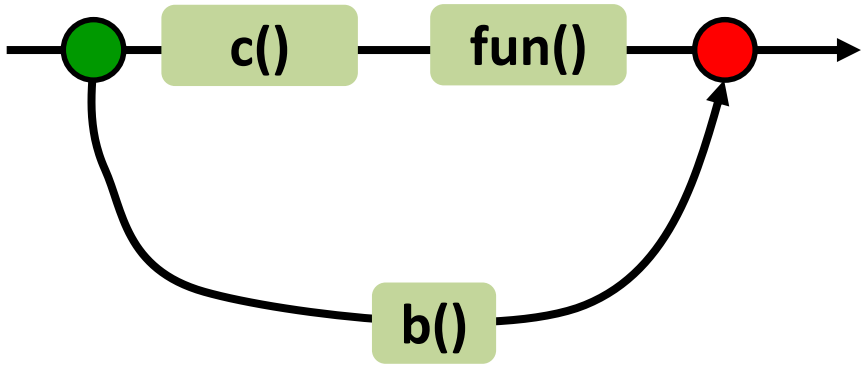
```
cilk_spawn b();
cilk_spawn c();
fun()
cilk_sync;
```



Если время работы fun() невелико, эффективнее объединить выполнение c() и fun() в одном strand

Версия 2

```
cilk_spawn b();
c();
fun()
cilk_sync;
```



MergeSort

```
// Merge sequences [xs, xe) and [ys, ye)
// to output [zs, (xe - xs) + (ye - ys)
void parallel_merge(T* xs, T* xe, T* ys, T* ye, T* zs) {
    const size_t MERGE_CUT_OFF = 2000;
    if (xe-xs + ye-ys <= MERGE_CUT_OFF ) {
        serial_merge(xs,xe,ys,ye,zs);
    } else {
        T *xm, *ym;
        if ( xe-xs < ye-ys ) {
           ym = ys + (ye - ys) / 2;
            xm = std::upper_bound(xs, xe, *ym);
        } else {
            xm = xs + (xe - xs) / 2;
            ym = std::lower_bound(ys, ye, *xm);
        }
        T* zm = zs + (xm-xs) + (ym-ys);
        cilk_spawn     parallel_merge(xs, xm, ys, ym, zs);
        /* no spawn */ parallel_merge(xm, xe, ym, ye, zm );
        // implicit cilk_sync
    }
}
```

cilk_for

- **cilk_for** сообщает компилятору, что итерации цикла можно выполнять параллельно
- В условии завершения цикло можно использовать только операции <, <=, >, >=, !=
- В блоке инкрементации управляющей переменной цикла допустимы операции: +=, -=, ++, --

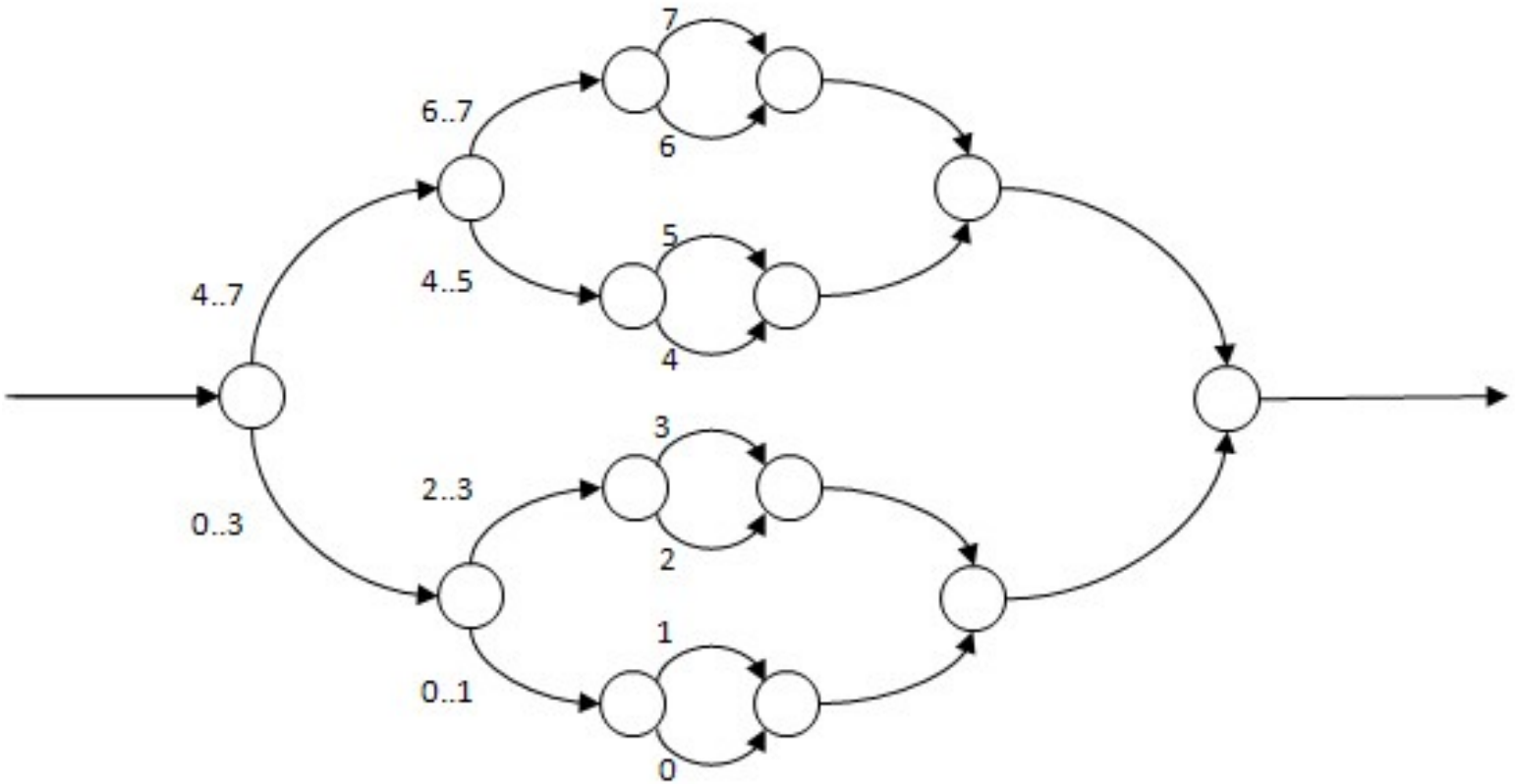
```
cilk_for(int i = begin; i < end; i += 2) {
    f(i);
}
```

```
cilk_for(T::iterator i(vec.begin()); i != vec.end(); ++i) {
    g(i);
}
```

cilk_for

- Компилятор Intel преобразует цикл в рекурсивную функцию с вызовами cilk_spawn (divide-and-conquer)

- **`cilk_for(i = 0; i < 8; ++i)`**
- **`grainsize = 1`** `// max количество итерация для strand`



cilk_for grainsize

- **grainsize** – максимальное количество итераций выполняемых нитью (strand)

- Параметр grainsize позволяет регулировать загрузку нитей

```
#pragma cilk grainsize = 4
cilk_for(i = 0; i < N; i++) {
    /* Code */
}
```

Intel Compiler: cilk_for

```
void run_loop(first, last)
{
    if (last - first) < grainsize) {
        for (int i = first; i < last; ++i)
            LOOP_BODY;
    } else {
        int mid = (last + first) / 2;
        cilk_spawn run_loop(first, mid);
        run_loop(mid, last);
    }
}
```

Пример SAXPY

```
void saxpy(float a, float *x, float *y, int n)
{
    for (int i = 0; i < n; i++) {
        y[i] += a * x[i];
    }
}
```

Пример SAXPY

```
void saxpy(float a, float *x, float *y, int n)
{
    cilk_for (int i = 0; i < n; i++) {
        y[i] += a * x[i];
    }
}
```

Reducers

- **Reducer** – это потокобезопасные глобальные переменные (реализованы на базе шаблонов C++)
- К такой переменной может быть применена групповая операция, формирующая итоговое значение по значениям из всех нитей (strands)
- Поддерживаемые операции:
  max, min, +, -, XOR, OR, AND, add, …
- Имеется возможность реализовывать пользовательские операции

Пример использования Reducers

```
unsigned int compute(unsigned int i)
{
    return i;
}

int main(int argc, char* argv[])
{
    unsigned long long int n = 1000000;
    unsigned long long int total = 0;

    for (unsigned int i = 1; i <= n; ++i) {
        total += compute(i);
    }

    return 0;
}
```

Пример использования Reducers: parallel

```
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>

unsigned int compute(unsigned int i)
{
    return i;
}

int main(int argc, char* argv[])
{
   unsigned long long int n = 1000000;
   // reducer с операцией +
   cilk::reducer_opadd<unsigned long long int> total(0);

   cilk_for (unsigned int i = 1; i <= n; ++i) {
     *total += compute(i);
   }

   std::cout << "Total " << total.get_value() << std::endl;
   return 0;
}
```

Reducers: lists

```
#include <cilk/reducer_list.h>

TreeNode *tree;

cilk::reducer_list_append<int> values;

void inorder_walk(TreeNode *node)
{
    if (node == NULL)
        return;
    cilk_spawn inorder_walk(node->left);
    values.push_back(node->value);
    inorder_walk(node->right);
}
```

Array Section Notation

- **Array Notation –** это языковые конструкции Intel Cilk Plus информирующие компилятор о возможности использовать векторные инструкции для обработки массивов (SIMD processing)

- **Обращение к части массива: base[first:length:stride]**
  - A[:] – весь массив
  - A[3:99] – элементы с 3 по 99
  - B[:][5] – столбец 5 матрицы
  - C[0:3][0:4] – подматрица

- **Вид векторных операций**

```
ArraySection2  OP  ArraySection2
```

Array Section Notation

```
s[0:n] = sin(y[0:n])
```

```
if (a[0:n] < b[0:n])
    c[0:n] += 1;
else
    d[0:n] -= 1;
```

```
/* Reductions */
sum = __sec_reduce_add(a[0:n] * b[0:n]);
```

#pragma simd

```
void saxpy(float a, float *x, float *y, int n)
{
    #pragma simd
    for (int i = 0; i < n; i++) {
        y[i] += a * x[i];
    }
}
```

#pragma simd

```
float sum(float *x, int n)
{
    float sum = 0;
    #pragma simd reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += *x++;
    }
    return sum;
}
```

Elemental functions

- **Elemental function** – это пользовательские C/C++-функции для реализации векторных операций (data parallelism)

```
// Elemental function
__attribute__((vector)) double ef_add(double x, double y)
{
    return x + y;
}

// Вызов функции с использованием array notations
a[:] = ef_add(b[:], c[:]);
a[0:n:s] = ef_add(b[0:n:s], c[0:n:s]);

// Вызов из cilk_for
_Cilk_for (j = 0; j < n; ++j)
{
  a[j] = ef_add(b[j],c[j])
}
```

Задание

- Прочитать про Holders (hyperobjects) в Intel Cilk Plus

- **Intel Cilk Plus** // Intel® C++ Compiler XE 13.1 User and Reference Guides, http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/index.htm

- Intel® Cilk™ Plus Language Extension Specification Version 1.2 (2013-09-06) http://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm

- Cilk lectures by Charles Leiserson and Bradley Kuszmaul (http://supertech.csail.mit.edu/cilk/)