Intel® Cilk™ Plus

С.А.Немнюгин
Санкт-Петербургский Государственный Университет

snemnyugin@mail.ru

Robotics & Embedded School - 2012

Многопоточная программа

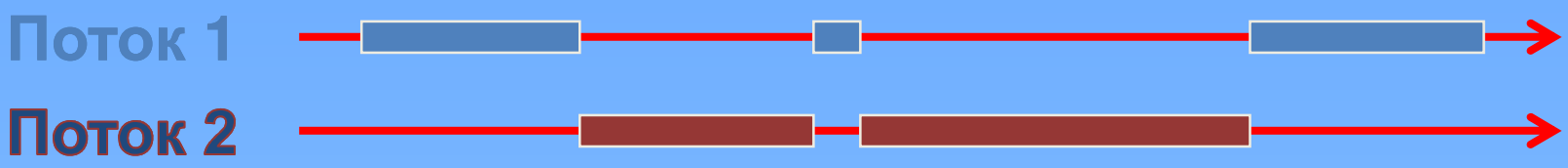
Поток (нить) представляет собой последовательный поток управления (последовательность команд) в рамках одной программы. При создании процесса порождается главный поток, выполняющий инициализацию процесса. Он же начинает выполнение команд.

Поток и процесс соотносятся следующим образом:

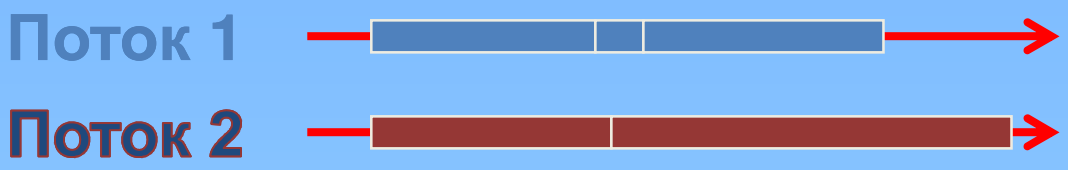
- с процессом ассоциирован главный поток, инициализирующий выполнение команд процесса;
- любой поток может порождать в рамках одного процесса другие потоки;
- каждый поток имеет собственный стек;
- потоки, соответствующие одному процессу, имеют общие сегменты кода и данных.

Конкуренция за ресурсы и параллельное исполнение

Конкуренция за ресурсы (видимый параллелизм)



Реальный параллелизм



Для реализации реального параллелизма требуется соответствующая архитектура – многоядерная или многопроцессорная с общей памятью.

При разработке многопоточных приложений возникают следующие проблемы:

- гонки за данными;
- блокировки;
- несбалансированность загрузки.

Гонки за данными (*data races*)

Гонки за данными являются следствием зависимостей, когда несколько потоков модифицируют содержимое одной и той же области памяти. Наличие гонок за данными не всегда является очевидным. Они могут приводить к конфликтам двух типов:

1) конфликт «чтение-запись»;
2) конфликт «запись-запись».

Борьба с гонками за данными:

- использование преимущественно локальных по отношению к потоку, а не разделяемых переменных;
- управление доступом к разделяемым переменным с помощью различных средств синхронизации (они могут быть реализованы с помощью семафоров, событий, критических секций, взаимных блокировок - мьютексов).

Блокировки

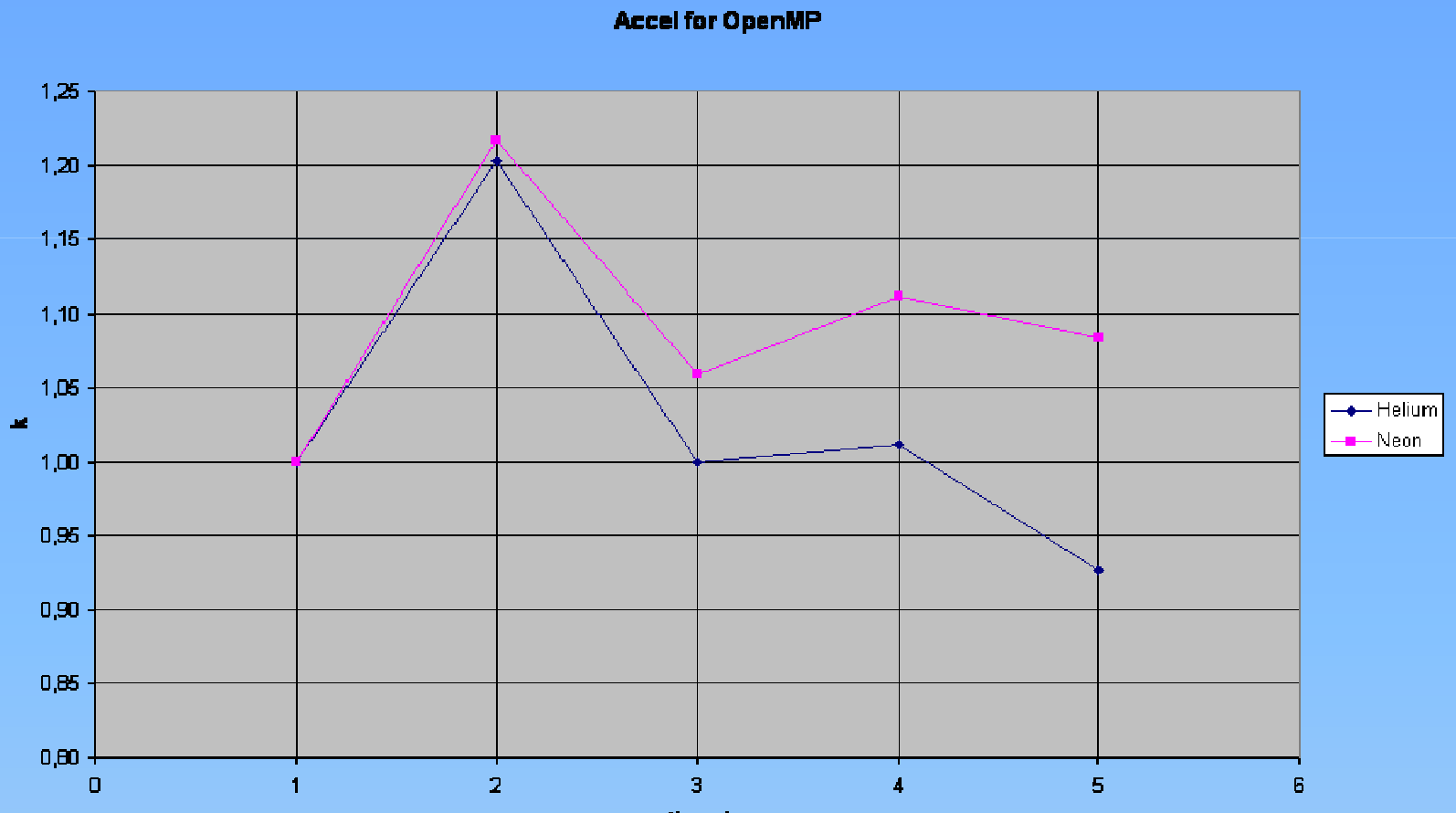
Блокировка (тупик) возникает, если поток ожидает выполнение условия, которое не может быть выполнено. Обычно возникновение тупиковой ситуации является следствием конкуренции потоков за ресурс, который удерживается одним из них.

Условия возникновения тупика

- доступ к ресурсу эксклюзивен (возможен только одним потоком);
- поток может удерживать ресурс, запрашивая другой;
- ни один из конкурирующих потоков не может освободить запрашиваемый ресурс.

Масштабируемость

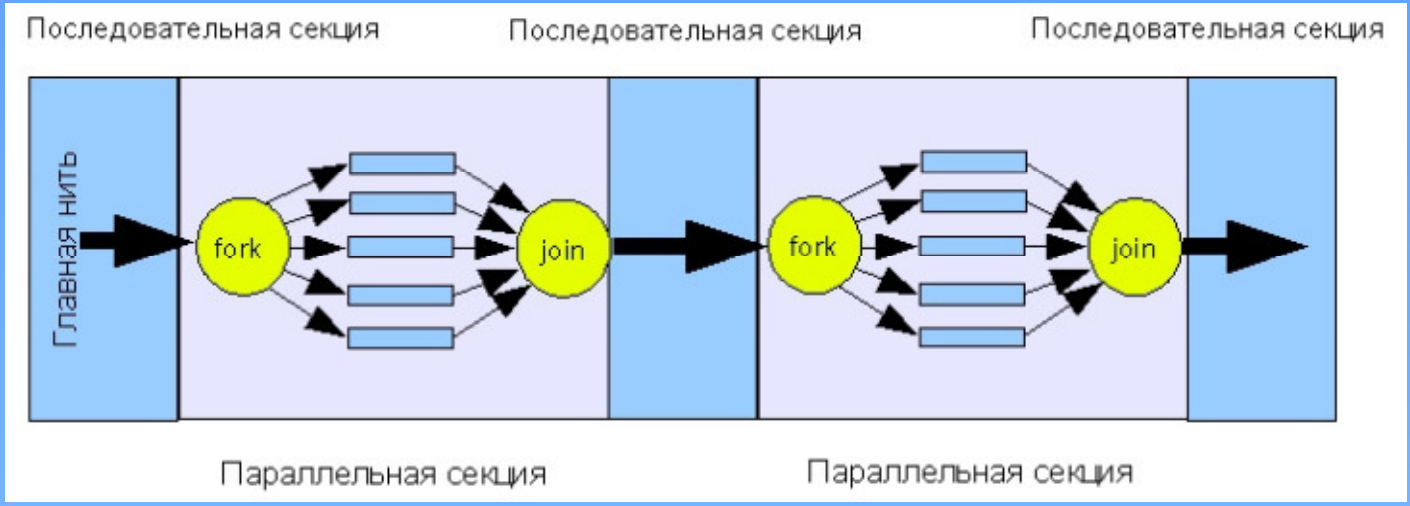
Число программных потоков должно совпадать с числом аппаратных потоков.
Зависимость ускорения от числа потоков для теста на 2-ядерной архитектуре:



Программные инструменты реализации многопоточного параллелизма

POSIX Threads **Windows API**	низкоуровневые инструменты
OpenMP **OpenCL** **Intel® Cilk™ Plus**	высокоуровневые инструменты

Программирование с OpenMP (Open MultiProcessing)



- программа состоит из последовательных и параллельных секций;
- в начальный момент времени порождается основная нить, выполняющая последовательные секции программы;
- при входе в параллельную секцию программы выполняется операция fork, порождающая набор нитей;
- каждая нить имеет свой уникальный числовой идентификатор (0 для мастер-нити). Все параллельные нити исполняют один код;
- при выходе из параллельной секции выполняется операция join, завершающая выполнение всех нитей кроме главной.

Pro

- возможность пошагового распараллеливания;
- переносимость;
- высокоуровневое программирование;
- поддержка языков Fortran и C/C++;
- поддержка модели параллелизма данных.

Contra

- масштабируемость ограничена архитектурой исполнения;
- программист должен думать не только о том, ЧТО должно выполняться параллельно, но и КАК;
- побочные эффекты использования глобальных переменных.

Структура

1) *Директивы компилятора* - используются для создания потоков, распределения работы между потоками и их синхронизации. Директивы включаются в исходный текст программы.

2) *Подпрограммы библиотеки времени выполнения* - используются для установки и определения атрибутов потоков. Вызовы этих подпрограмм включаются в исходный текст программы.

3) *Переменные окружения* - используются для управления поведением параллельной программы.

Привязка к C/C++

В программах на языке C прагмы, имена функций и переменных окружения OpenMP начинаются с omp, omp_ или OMP_. Формат директивы:

```
#pragma omp директива [оператор_1[, оператор_2, :]]
```

В OpenMP-программе используется заголовочный файл omp.h.

Привязка к языку Fortran

В программах на языке Fortran директивы компилятора, имена подпрограмм и переменных окружения начинаются с OMP или OMP_. Формат директивы компилятора:

```
{!|C|*}$OMP директива [оператор_1[, оператор_2, :]]
```

Директива начинается в первой (фиксированный формат записи текста языка Fortran 77) или произвольной (свободный формат) позиции строки. Допускается продолжение директивы в следующей строке, в этом случае действует стандартное в данной версии языка правило для обозначения строки продолжения (непробельный символ в шестой позиции для фиксированного формата записи и амперсанд для свободного формата).

Пример

```
#include "omp.h"
#include <stdio.h>

double f(double x)
{
return 4.0 / (1 + x * x);
}
main ()
{
const long N = 100000;
long i;
double h, sum, x;
sum = 0;
h = 1.0 / N;

#pragma omp parallel shared(h)
{
#pragma omp for private(x) reduction(+:sum)

for (i = 0; i < N; i++)
{
x = h * (i + 0.5);
sum = sum + f(x);
}
}
printf("PI = %f\n", sum / N);
}
```

Эффективность

Эффективность приложения, распараллеленного с помощью OpenMP, зависит от баланса между выигрышем от распараллеливания и накладными расходами на организацию многопоточности, диспетчеризацию, синхронизацию и т.д.

Накладные расходы

parallel	1.5 мкс (Intel ® Xeon 3Ггц)
barrier	1.0
schedule(static)	1.l0
schedule(guided)	6.0
schedule(dynamic)	50.0
ordered	0.5
single	1.0
reduction	2.5

Диспетчеризацией параллельной OpenMP-программы управляет программист с помощью оператора schedule. Поддерживаются три способа распределения работы между потоками: статический, динамический и «управляемый».

Intel® Cilk™ Plus – общая характеристика

Intel® Cilk™ Plus

- расширение языков C/C++ (ключевые слова, расширенная векторная нотация, гиперобъекты, элементные функции);
- обеспечивает эффективный и безопасный параллелизм типа «fork-join» (операция порождения – spawn, гиперобъекты, диспетчеризация системой исполнения);
- обеспечивает векторный параллелизм (векторизация операций с сечениями массивов и элементных функций);
- Fortran (в настоящее время) не поддерживается.

Intel® Cilk™ Plus поддерживается компиляторами:

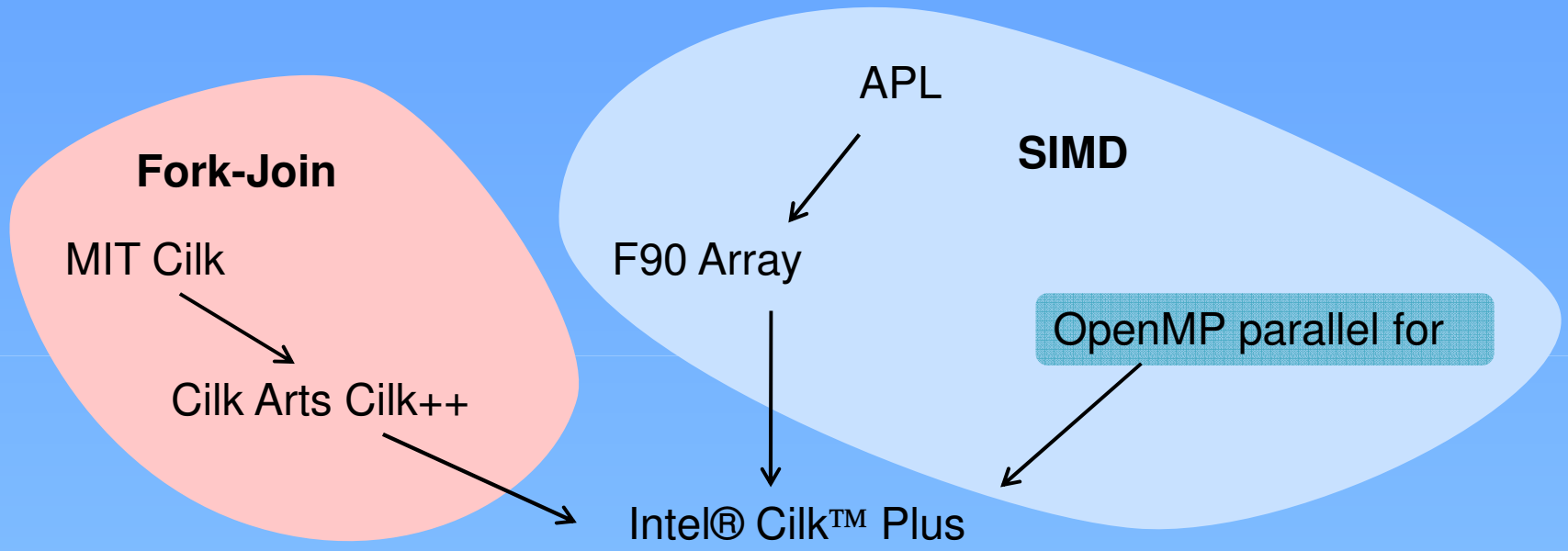
- Intel (начиная с версии 12);
- GCC (начиная с версии 4.7);
- совместим с Microsoft Visual Studio.

Мультиплатформенность (Windows и Linux).

Ориентирован на «обычные» процессоры Intel, а также на ускоритель MIC (Many-Integrated-Core). GPGPU (графические ускорители) не поддерживаются.

Хронология событий

1992 – теория планировщика Cilk [Blumofe и Leiserson]
1993 – первая реализация Cilk на CM-5.
1995 – окончательная версия Cilk. [Blumofe и др.]
1998 – современная реализация Cilk. [Frigo и др.]
1998 – детектор гонок за данными. [Feng и Leiserson]
2004 – адаптивная диспетчеризация. [Agrawal]
2005 – JCilk. [Danaher и др.]
2007 – основана Cilk Arts. [Frigo и Leiserson]
2007 – появление Cilk++ и его реализации.
2009 – Cilk Arts приобретён Intel.
2010 – появление Cilk Plus: индексная нотация и интеграция с инструментами , Intel.


Fork-Join
MIT Cilk
Cilk Arts Cilk++
APL
SIMD
F90 Array
OpenMP parallel for
Intel® Cilk™ Plus

Удобные средства работы с массивами (расширенная индексная нотация – аналог сечений массивов в языке Fortran).

Удобное использование векторных расширений команд, векторизация функций.

В **Intel® Cilk™ Plus** сохраняется семантика последовательной программы.

Программа может выполняться как в последовательном, так и в параллельном режимах.

Параллельное выполнение возможно, если это допускает целевая платформа (достаточное количество ядер).

Пример

```
#include <cilk/cilk.h>
 void sample_qsort(int * begin, int * end)
{
        if (begin != end) {
        --end;
        int * middle = std::partition(begin, end,
                std::bind2nd(std::less<int>(),*end));
        std::swap(*end, *middle); // pivot to middle
        cilk_spawn sample_qsort(begin, middle);
        sample_qsort(++middle, ++end); // Exclude pivot
        cilk_sync;
,       }
}
```

Модель программирования Intel® Cilk™ Plus

Модель программирования **Intel® Cilk™ Plus** основана на *параллелизме задач*.
Программа пишется в семантике последовательного программирования. Фрагменты для распараллеливания расщепляются на подзадачи, связанные отношениями подчинения («родитель»-«потомок»). Такая реализация параллелизма иногда называется «fork-join».

Программист, использующий **Cilk™ Plus** должен думать о том, ***что*** следует распараллелить, а не ***как***. В этом – одно из отличий от OpenMP-программирования.

Балансировкой занимается runtime-система. Балансировка выполняется методом *захвата работы*. Алгоритмы диспетчеризации таковы, что их эффективность, как правило, высока.

Программист

Определяет и описывает потенциальный параллелизм.

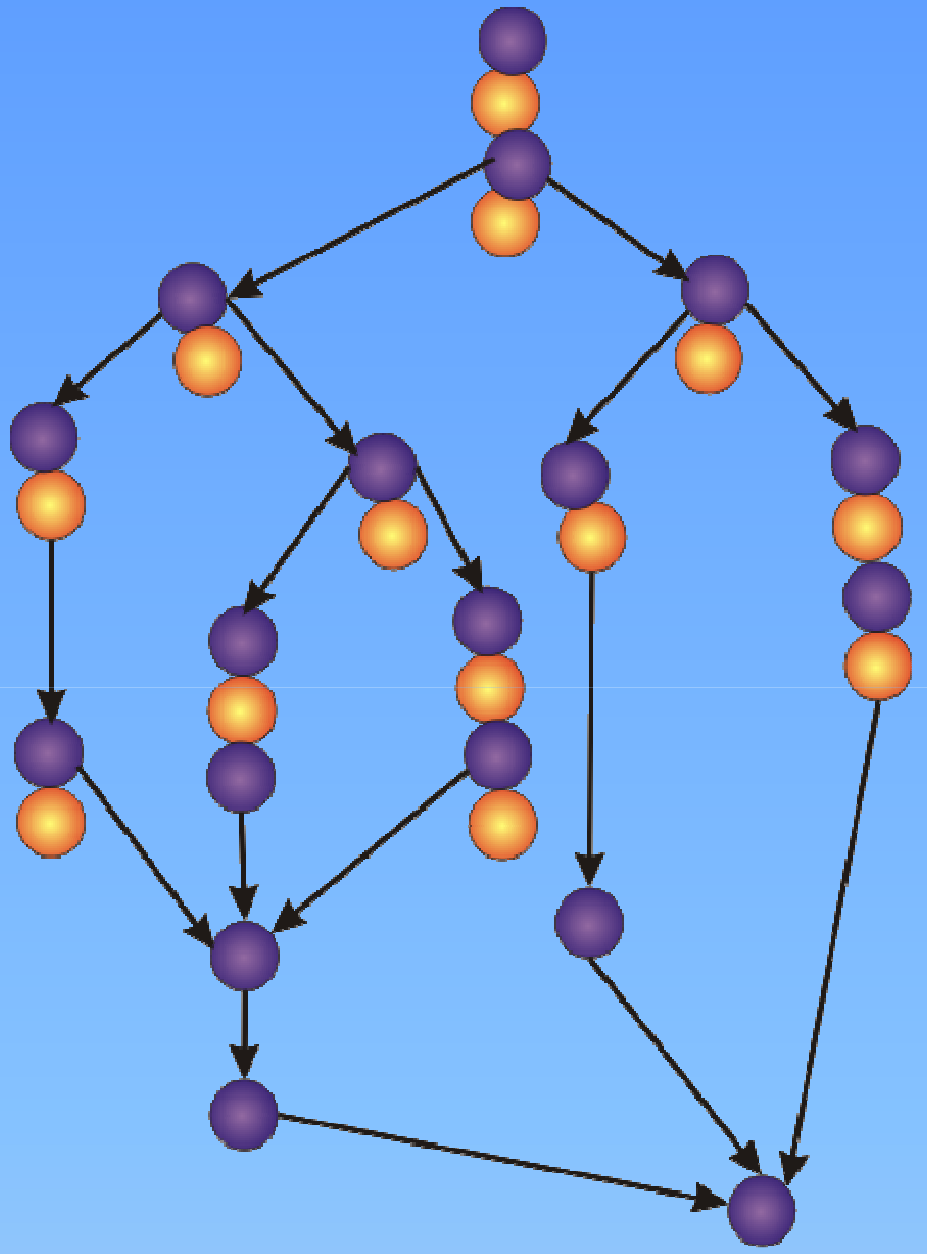
Планировщик

Отображает его на реально существующую конфигурацию потоков.

Задачи связаны между собой отношениями подчинения. Конфигурацию приложения во время его выполнения можно изобразить в виде направленного ациклического графа (DAG).

Граф задач в Cilk-программе является динамическим – он создаётся и изменяется в процессе выполнения программы.

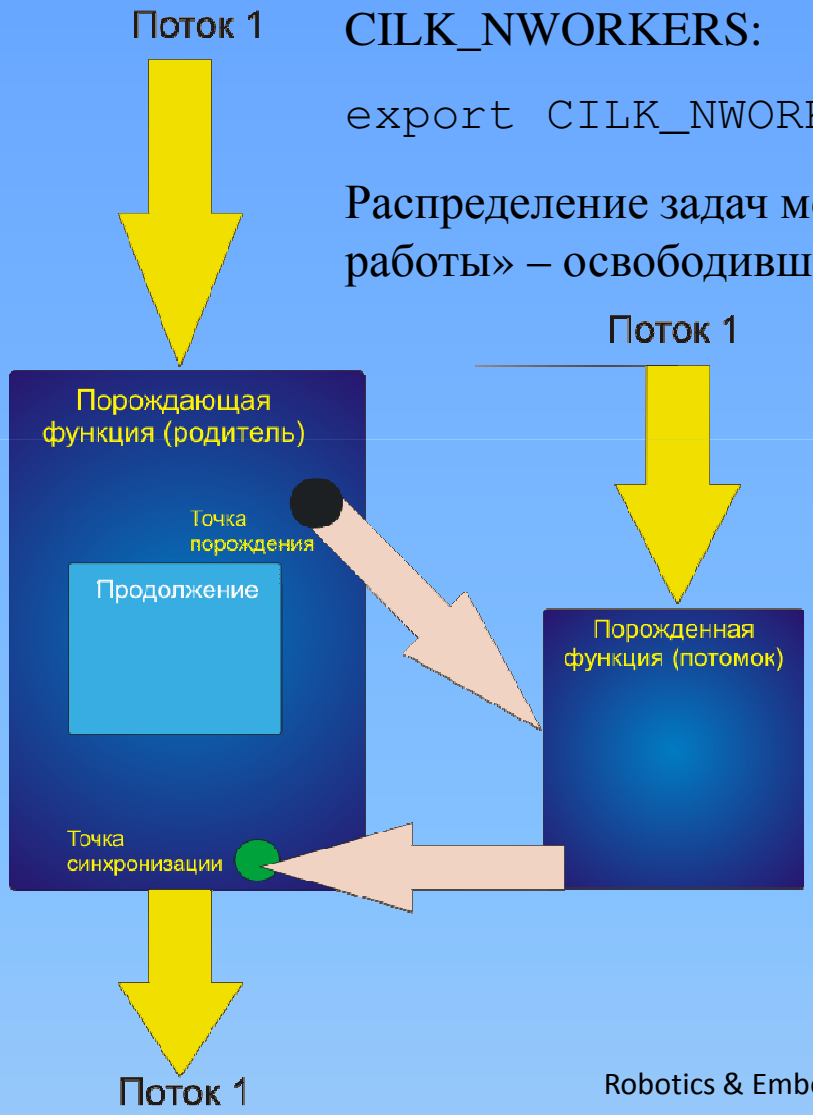
- «Ветви» - последовательные фрагменты кода. Исполняются в режимах «продолжения» и «порождения».
- Узлу порождения соответствуют 2 «наследника».



При выполнении параллельной Cilk-программы формируется очередь задач. Выполнением задач занимаются «исполнители» (workers). Это- потоки.
Их число задаётся с помощью переменной окружения CILK_NWORKERS:

```
export CILK_NWORKERS=4 (Linux/bash)
```

Распределение задач между потоками выполняется методом «захвата работы» – освободившийся поток выполняет очередную задачу.



Если доступен только один поток, программа выполняется как последовательная

Диспетчер использует распределённый алгоритм захвата работы. Это «жадный» алгоритм.

Сериализация (выполнение программы в последовательном режиме) происходит, если степень параллелизма целевой платформы недостаточно велика.

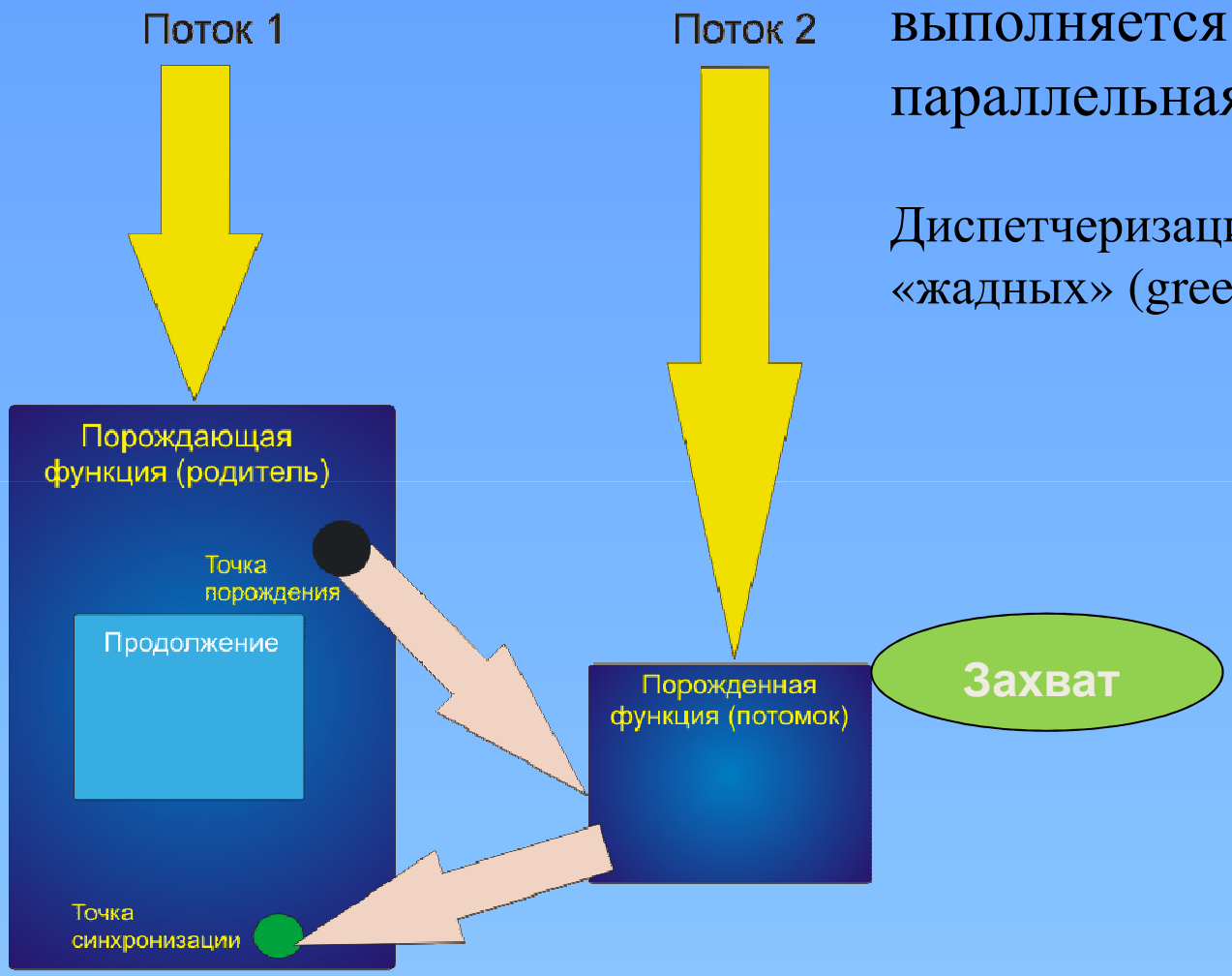
Сериализация также происходит при использовании заголовочного файла <cilk/cilk_stub.h> и при компиляции с соответствующим ключом:

icc: -cilk-serialize
icl: /Qcilk-serialize

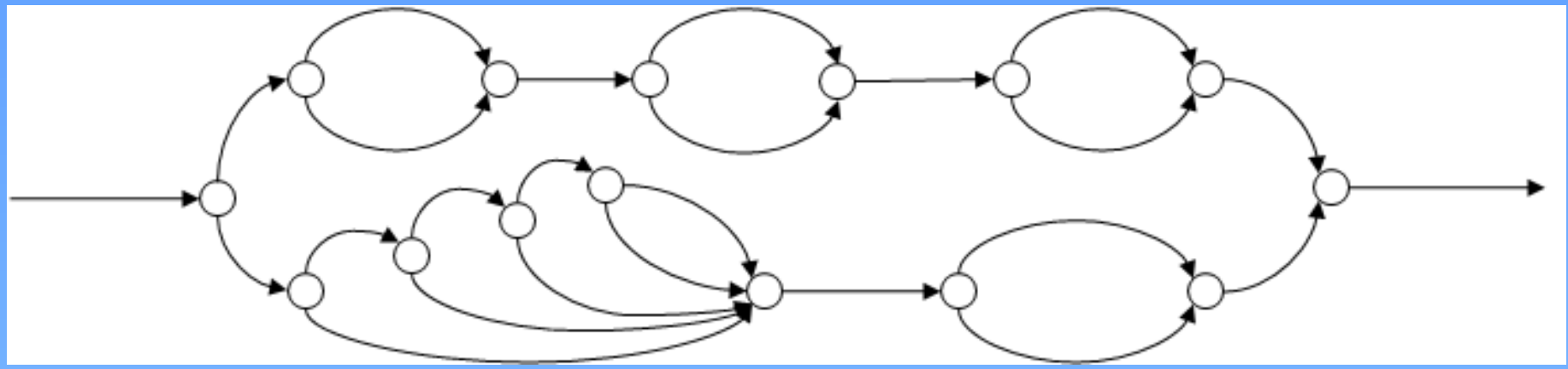
В Microsoft Visual Studio сериализовать Cilk-программу можно так:
Properties → C/C++ → Language [Intel C++] → Replace Intel Cilk Plus Keywords with Serial Equivalent

Если доступно несколько потоков, программа выполняется как параллельная.

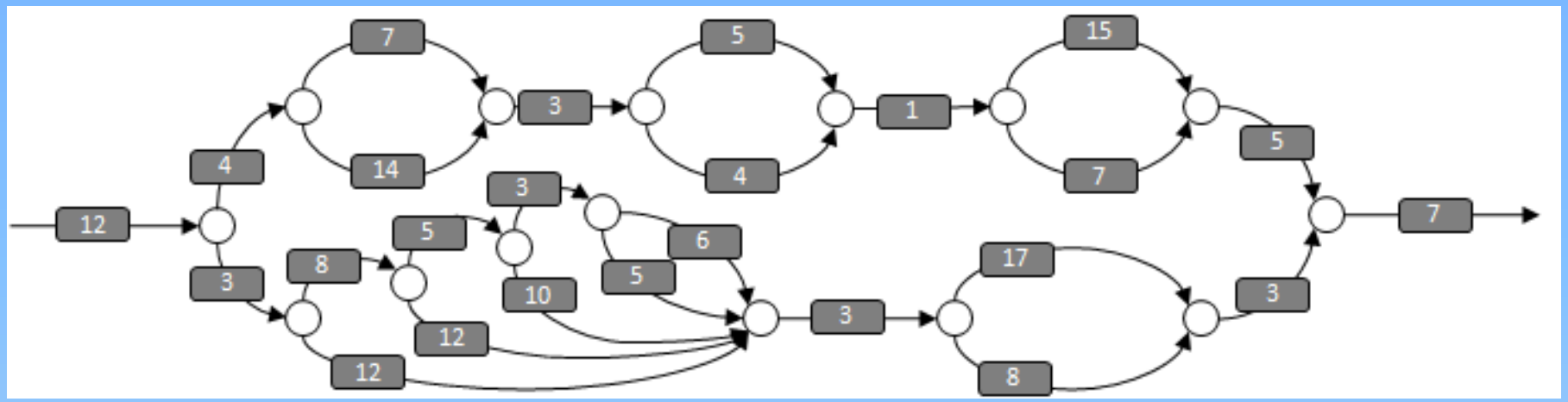
Диспетчеризация основана на «жадных» (greedy) алгоритмах.



Cilk-программа во время её исполнения может быть представлена следующим динамическим ациклическим графом:



Пусть время исполнения различных ветвей Cilk-программы такое, как на диаграмме:

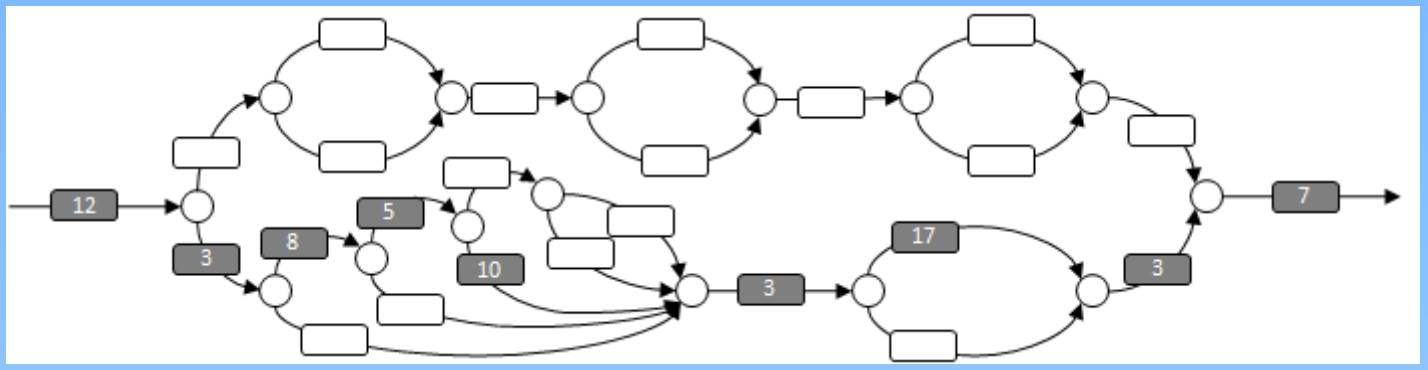


Работа (work) - суммарный объём процессорного времени по всем веткам графа программы. Это время выполнения Cilk-программы с одним исполнителем (T_1).

В примере это 181 мс.

Длина критического пути (span) – наиболее «длинный» путь в графе от начала программы к её концу. Это время выполнения параллельной Cilk-программы, если есть неограниченное число исполнителей (T_∞).

В примере это 68 мс.



Примеры оценок work и span для некоторых алгоритмов

	Work	Span
Merge sort	$\Theta(n \lg n)$	$\Theta(\lg^3 n)$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\lg n)$
Strassen	$\Theta(n^{\lg 7})$	$\Theta(\lg^2 n)$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \lg n)$
Tableau construction	$\Theta(n^2)$	$\Omega(n^{\lg 3})$
FFT	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$
SpMV (CSR)	$\Theta(nnz)$	$\Theta(\lg n)$
SpMV & SpMV_T (CSB)	$\Theta(nnz)$	$\Theta(\sqrt{nnz} \lg n)$
Breadth-first search	$\Theta(E)$	$\Theta(d \lg V)$

Время исполнения в условиях реального параллелизма –время выполнения параллельной Cilk-программы на P процессорах (T_P).

Ускорение = T_1 / T_P

Закон 1

$T_P >= T_1 / P$

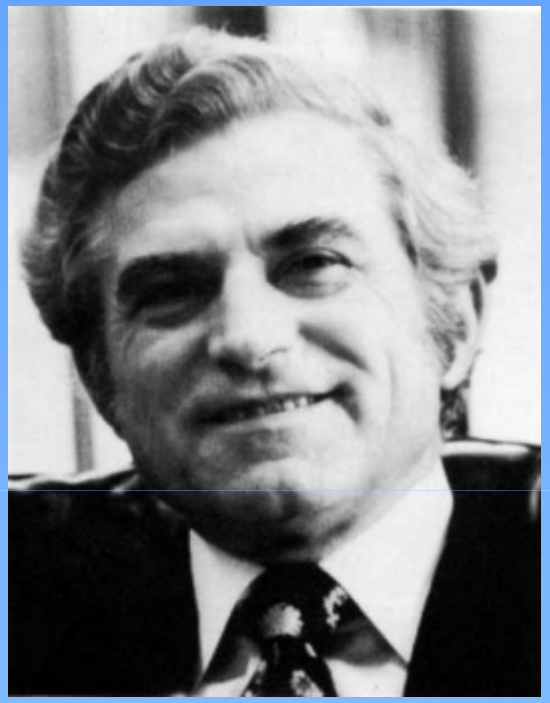
Закон 2

$T_P >= T_\infty$

Закон максимального ускорения

Ускорение <= T_1 / T_∞

T_1 / T_∞ - параллелизм



Дж.Амдал

Теорема

Диспетчер Cilk достигает времени выполнения

$T_P = T_1 / P + O(T_\infty)$

Доказательство

?

Следствие 1

Любой диспетчер, работающий на основе «жадного» алгоритма уступает в эффективности оптимальному не более чем в 2 раза.

Следствие 2

Любой диспетчер, работающий на основе «жадного» алгоритма позволяет достичь линейного ускорения, если $T_1 / T_\infty >> P$.

Структура Intel® Cilk™ Plus

Ключевые слова (всего 3!):

- `cilk_spawn` – порождение задачи;
- `cilk_for` – распараллеливание цикла;
- `cilk_sync` – синхронизация задач.

Низкие накладные расходы.

Гиперобъекты (редукторы)

Редукторы – «параллельные» глобальные переменные, позволяющие избежать гонок за данными и блокировок.

Эффективное управление редукторами обеспечивается системой исполнения Cilk-программ.

Функции прикладного программного интерфейса (API)

- `__cilkrts_set_param("nworkers", "4")`
- `__cilkrts_get_nworkers()`
- `__cilkrts_get_total_workers()`
- `__cilkrts_get_worker_number()`

Расширенная индексная нотация

Отличает **Cilk™ Plus** от **Cilk™.**

Удобная запись операций с массивами.

Более высокая эффективность операций с массивами (компилятор порождает исполняемый код, использующий векторные инструкции).

Сходство с сечениями массивов языка Fortran, при различии в синтаксисе и реализации.

```
if (a[:] > b[:]) {
c[:] = d[:] * e[:];
} else {
c[:] = d[:] * 2;
}
```

Элементные (векторные) функции

Элементные функции обеспечивают векторизацию вычисления математических функций.

Аргументы элементных функций – векторы значений.

Возвращается вектор результата, конформный векторному аргументу.

Пример:

```
__declspec (vector) double heat_distribution(
double XL, double YL, double t, double sigma,
double time)
{
double time_sqrt = sqrt(time);
double d1 =
(log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*
time_sqrt;
double d2 = d1-(sigma*time_sqrt);
return S*N(d1) - K*exp(-t*r)*N(d2);
}
```

Директива `SIMD`

Векторизация тех фрагментов кода, которые необходимо векторизовать – подсказка векторизующему компилятору.

Используются векторные регистры (AVX) процессора.

Используется векторное расширение команд процессора SSE (Streaming SIMD Extension).

Компилятор генерирует векторизованный код.

Программист гарантирует корректность векторной семантики.

Циклы остаются без изменений.

Пример

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

Файл заголовков

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

Переменные окружения

Основная - `CILK_NWORKERS` (количество потоков).

Ключевые слова

Ключевое слово `cilk_spawn`

`cilk_spawn` является псевдонимом для `_Cilk_spawn`.

Обозначает *точку порождения*. В этой точке создаётся новая задача, выполнение которой может быть продолжено данным потоком или захвачено другим (параллельным) потоком. Ключевые слова только обозначают место в программе, где возможен (**но не обязателен!**) параллелизм.

`cilk_spawn` является указанием системе исполнения на то, что данная функция может (но не обязана) выполняться параллельно с функцией, из которой она вызвана.

Синтаксис (допустим любой из трёх):

```
cilk_spawn имя_функции_потомка()
type var = cilk_spawn имя_функции_потомка()
var = cilk_spawn имя_функции_потомка()
```

Допускается:

```
var = cilk_spawn (object.*pointer)(args);

cilk_spawn [&]{ g(f()); }();

cilk_spawn g(f());
```

Два последних варианта – в первом обе функции выполняются в потомке, во втором случае сначала выполняется `f()`, затем – потомок.

Порождение с использованием лямбда-выражения:

```
cilk_spawn [&]{
    for( int i=0; i<n; ++i )
        a[i] = 0;
} ();
...
cilk_sync;
```

Не допускается:

```
g(cilk_spawn f());
```

Пример:

```
void floyd_warshall() {
      for (int k = 0; k < n; ++k)
            for (int i = 0; i < n; ++i)
                  cilk_spawn work(k,i);
}
```

Ключевое слово `cilk_sync`

`cilk_sync` является псевдонимом для `_Cilk_sync`.

Обозначает *точку синхронизации*. В этой точке выполнение задач синхронизируется (барьерная синхронизация).

Выполнение функции с точкой синхронизации невозможно параллельно с потоком. Оно приостанавливается до тех пор, пока не будет завершён потомок. Затем выполнение функции возобновляется.

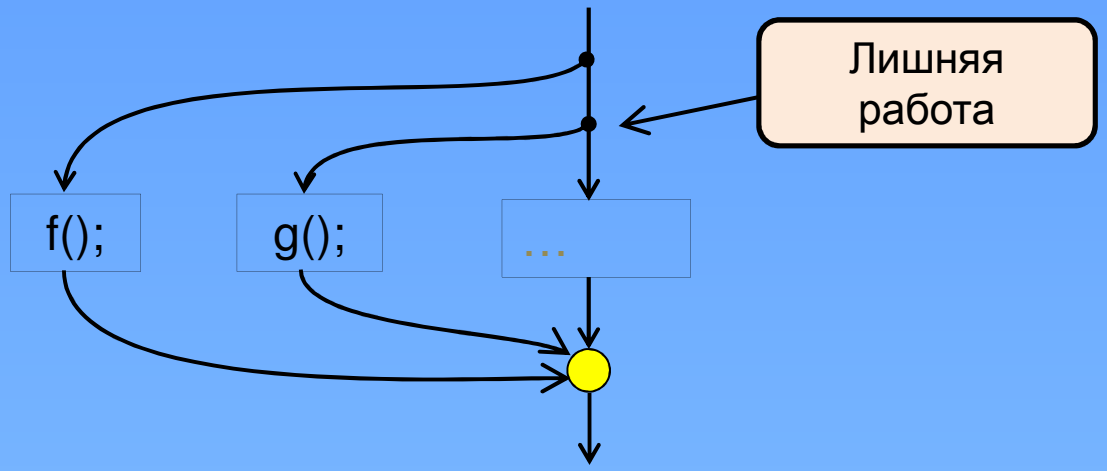
Неявно точка синхронизации присутствует в конце каждой функции.

Пример:

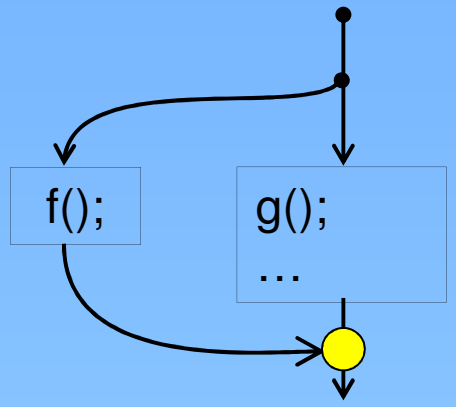
```
public:
    CilkSpawnSum(int nThreads) : _nThreads(nThreads),
result(0) {}
    virtual double FindSum(SimpleArray &data) {
        for(int i=0; i<_nThreads; i++) {
            cilk_spawn _SumComputer(data,
                i * data.GetSize() / _nThreads,
                (i+1) * data.GetSize() / _nThreads);
        }
        cilk_sync;
        return result.get_value();
    }
```

Плохой и хороший стиль

```
// Плохой стиль
cilk_spawn f();
cilk_spawn g();
...
cilk_sync;
```




```
// Хороший стиль
cilk_spawn f();
g();
...
cilk_sync;
```



Ключевое слово `cilk_for`

`cilk_for` является псевдонимом к `_Cilk_for`.

Распараллеливание цикла. В программе используется вместо заголовка цикла с параметром:

```
cilk_for(int k = 0; k < Niterations; ++k) {тело цикла}
```

В конце цикла используется барьерная синхронизация – исполнение программы продолжается только после завершения всех итераций.

Синтаксис (допустим любой из трёх):

```
cilk_for(описания; условное выражение; приращение)
```

Ограничения

- Распараллеливаются только циклы без цикловых зависимостей (итерации могут выполняться независимо).
- Недопустимы переходы в тело цикла и из него (операторы `return`, `break`, `goto` с переходом из тела цикла или в тело цикла).
- В цикле должна быть только одна переменная цикла.
- Переменная цикла не должна модифицироваться в цикле.
- Границы изменения параметра и шаг не должны меняться в теле цикла.
- Цикл не должен быть бесконечным.

Пример:

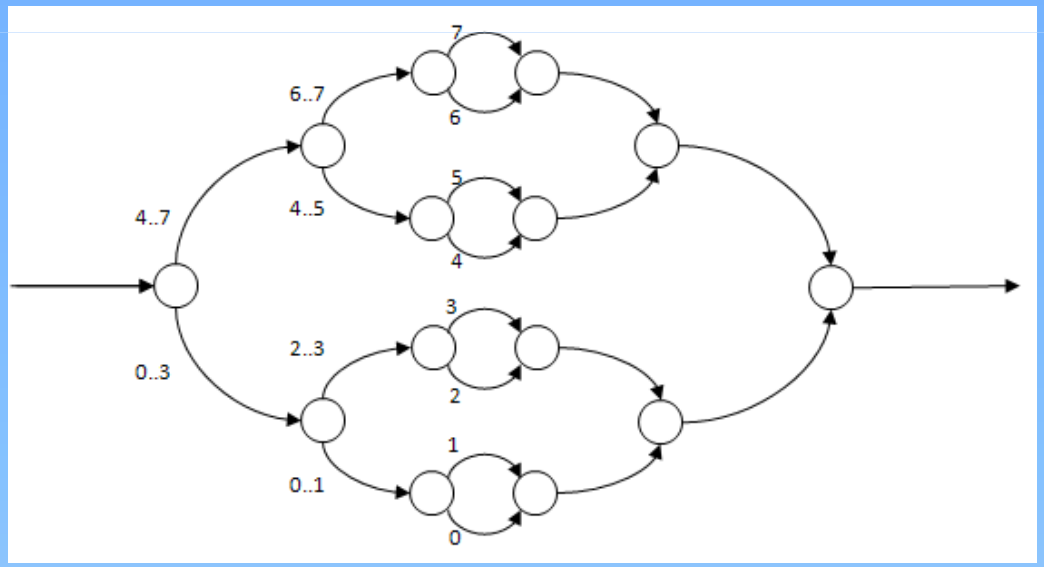
```
void floyd_warshall() {
     cilk_for (int k = 0; k < n; ++k)
          for (int i = 0; i < n; ++i)
               work(k,i);
}
```

Алгоритм

Рекурсивный алгоритм divide-and-conquer.

Нельзя считать, что каждая итерация цикла запускается в параллельном потоке!

Компилятор преобразует тело цикла в функцию, которая вызывается рекурсивно, с использованием стратегии «разделяй и властвуй».



На каждом уровне рекурсии половина оставшейся работы выполняется потомком , а вторая половина – продолжением.

Такой алгоритм позволяет обеспечить оптимальный баланс накладных расходов и выигрыша в результате распараллеливания для циклов с разной сложностью.

```
#pragma cilk grainsize = 1
cilk_for (int Niterations = 0; Niterations < 8; ++ Niterations)
f(Niterations);
```

Ключевое слово `grainsize` задаёт зернистость распараллеливания (количество итераций в наименьшей «порции», которые будут выполняться последовательно).

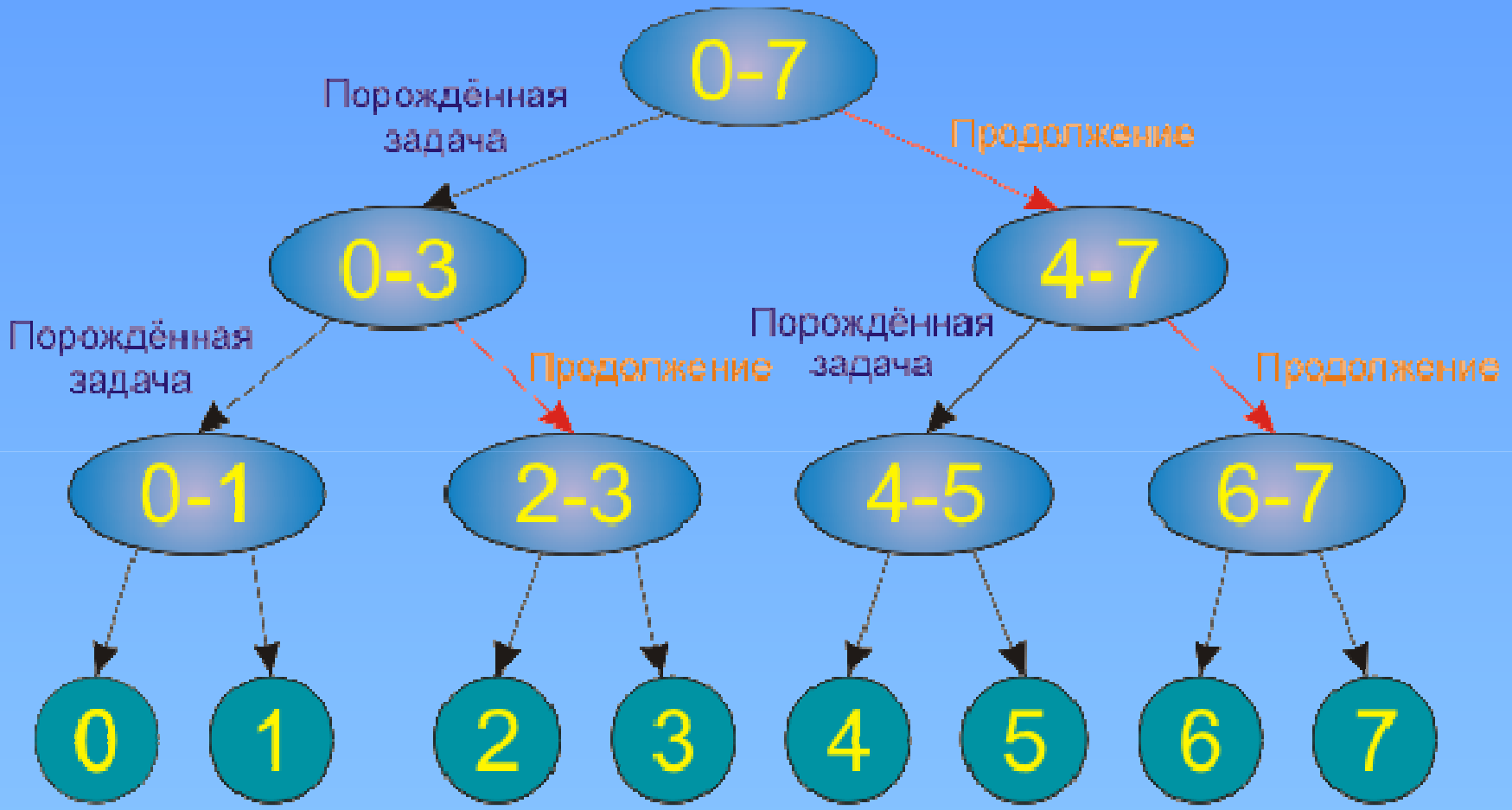
Если зернистость не указано явно, используется следующая формула:

```
#pragma cilk grainsize = min(512, N / (8*p))
```

Здесь N – число итераций цикла, p – число исполнителей. В случае, когда N>4096*p, зернистость устанавливается равной 512.

Если grainsize = 0, используется формула по умолчанию.
Если grainsize < 0, результат не определён.

Если
```
#pragma cilk grainsize = n/(4*__cilkrts_get_nworkers())
```
Зернистость будет определяться во время выполнения программы.


0-7
Порождённая задача
Продолжение
0-3
4-7
Порождённая задача
Продолжение
Порождённая задача
Продолжение
0-1
2-3
4-5
6-7
0
1
2
3
4
5
6
7

Как выбрать оптимальное значение зернистости

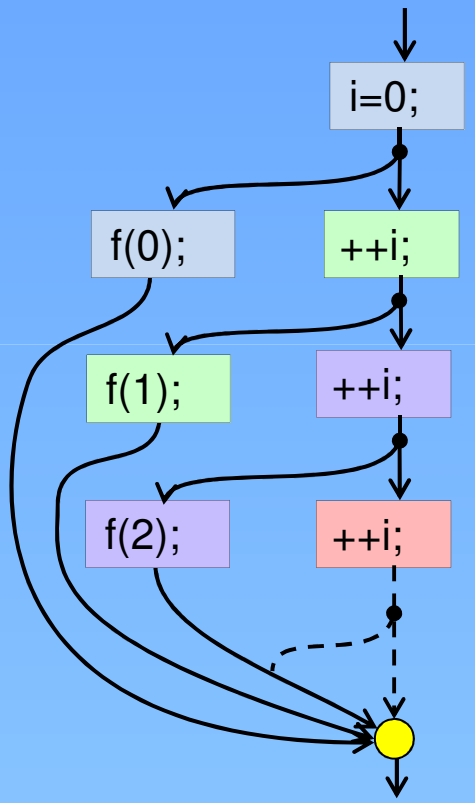
Если количество работы значительно варьируется от итерации к итерации, следует уменьшить grainsize.

Если все итерации «маленькие» (в смысле вычислительной сложности) , следует увеличить grainsize.

Оптимальность выбора grainsize следует подтверждать опытнм путём!

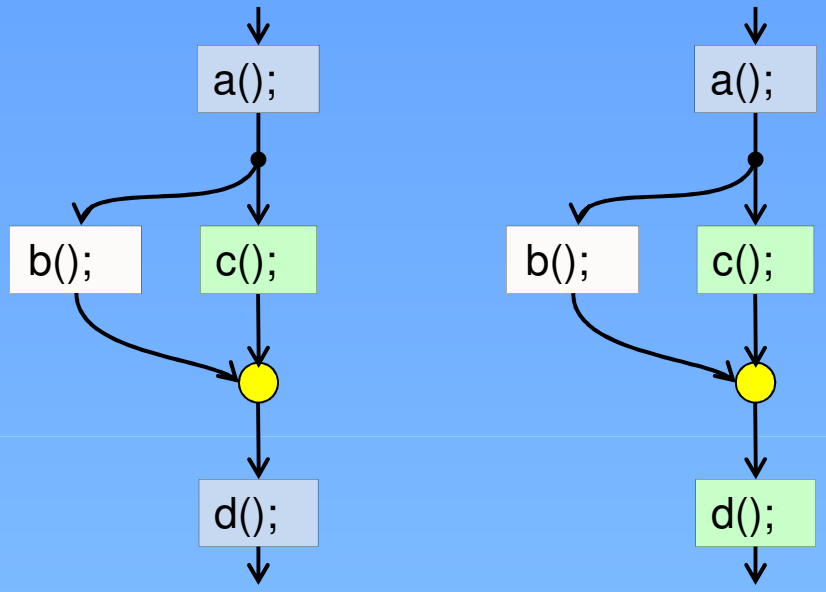
Пример

```
for( int i=0; i<n;
++i )
    cilk_spawn f(i);
cilk_sync;
```



Пример

```
a();
cilk_spawn
b();
c();
cilk_sync;
d();
```



Пример

```
…
#include <cilk/cilk.h>

int const n = 500;
int dist[n][n];

void work(int k, int i)
{
       for (int j = 0; j < n; ++j)
              if ((dist[i][k] * dist[k][j] != 0) && (i != j))
                     if ((dist[i][k] + dist[k][j] < dist[i][j]) ||
                            (dist[i][j] == 0))
                                   dist[i][j] = dist[i][k] +
dist[k][j];
}

void floyd_warshall() {
       cilk_for (int k = 0; k < n; ++k)
              for (int i = 0; i < n; ++i)
                     work(k,i);
}
```

```
int main(int argc, char *argv[]) {
	srand(time(NULL));
	int i, j;
	for (i = 0; i < n; ++i)
		for (j = 0; j < n; ++j)
		{
			if(i!=j)
				dist[i][j]=rand()%130;
			else
				dist[i][j]=0;
		}
	floyd_warshall();
}
```

Накладные расходы на диспетчеризацию

Сравним два варианта распараллеливания двойного цикла:

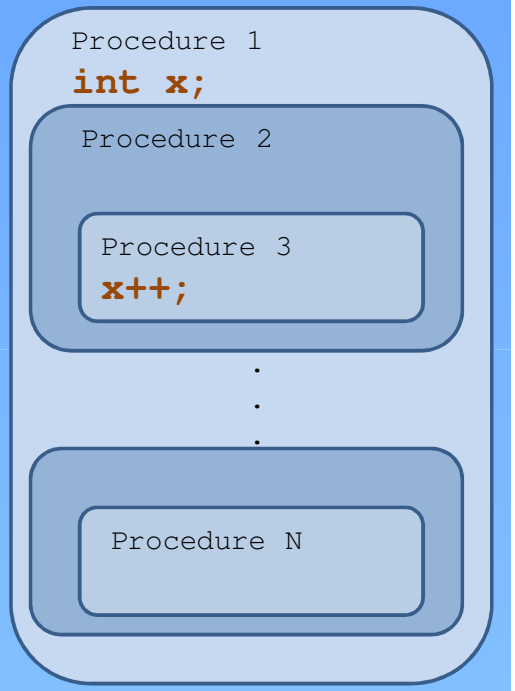
```
// A
cilk_for (int i = 0; i < 4; ++i)
    for (int j = 0; j < 1000000; ++j)
        do_work();

// B
for (int j = 0; j < 1000000; ++j)
    cilk_for (int i = 0; i < 4; ++i)
        do_work();
```

Эффективность распараллеливания фрагмента A выше, чем эффективность распараллеливания фрагмента B.

Область видимости переменных в многопоточных программах. Проблемы и решения

Область видимости – один из важнейших атрибутов переменной. Если область видимости ограничена, переменная называется *локальной*. Если область видимости переменной совпадает с программой, переменная называется *глобальной*.

В многопоточном программировании область видимости получает дополнительное измерение – видимость между потоками. Исполнение параллельной программы перестаёт быть локальным!



Pro

Использование глобальных переменных позволяет избежать «раздувания» списков параметров. Глобальными объявляются часто используемые параметры.

Contra

Побочные эффекты использования глобальных переменных могут препятствовать эффективной реализации параллелизма.

Пусть два оператора из разных потоков имеют доступ к одной переменной `x`. Тогда могут существовать 3 типа гонок за данными:

Оператор 1	Оператор 2	Тип «гонок за данными»
чтение	чтение	отсутствуют
чтение	запись	по чтению
запись	чтение	по чтению
запись	запись	по записи

Размер машинного слова может влиять на наличие гонок – при работе с упакованными структурами данных. Пример:

```
struct{
    char a;
    char b;
} x;
```

Наличие гонок может зависеть от уровня оптимизации.

Побочные эффекты

Гонки за данными – возникают при одновременном доступе из разных потоков к одной переменной. Отрицательный эффект – утрата детерминизма в поведении программы, утрата корректности. Это происходит, если:

- хотя бы один поток производит запись в общую переменную,;
- доступ к переменной происходит одновременно.

Как избежать гонок за данными

Синхронизация доступа к переменной.

Использование **локальных** относительно потоков переменных.

Гиперобъекты в Intel® Cilk™ Plus

Гиперобъекты (**редукторы**) в Intel® Cilk™ Plus – реализуют механизм разрешения гонок за данными.

Гиперобъект (редуктор) – в простейшем случае объект, с которым ассоциированы: значение, начальное значение, функция приведения.

Обращаться с редуктором надо как с объектом. Например, запрещено прямое копирование – результат такого копирования не определён.

При работе с редукторами не надо использовать блокировки => увеличивается производительность.

Редукторы сохраняют последовательную семантику: результат параллельной программы совпадает с результатом последовательной программы – при этом не требуется реструктуризация кода.

Редукторы можно использовать не только в циклах.

Переменная может быть описана как *редуктор* относительно ассоциативной операции (сложение, умножение, логическое И, объединение списков и другие). Требуется также использование соответствующего заголовочного файла. Пример:

Гиперобъекты эффективно реализованы – уменьшение непроизводительных расходов на синхронизацию доступа.

Вопрос – эффективность гиперобъектов при работе на виртуальных машинах.

Редукторы описываются как шаблоны C++ с описанием интерфейса с системой исполнения.

Редуктор – гиперобъект, с которым ассоциировано значение, операции инициализации и приведения.

Редукторы нельзя копировать напрямую, использование `memcpy()` приведет к непредсказуемым результатам. Следует использовать конструктор.

Операция должна быть ассоциативной, в противном случае не гарантируется детерминизм.

Следует помнить, что операции с плавающей точкой в любом случае могут утрачивать ассоциативность. Корректность программы в этом случае следует проверять.

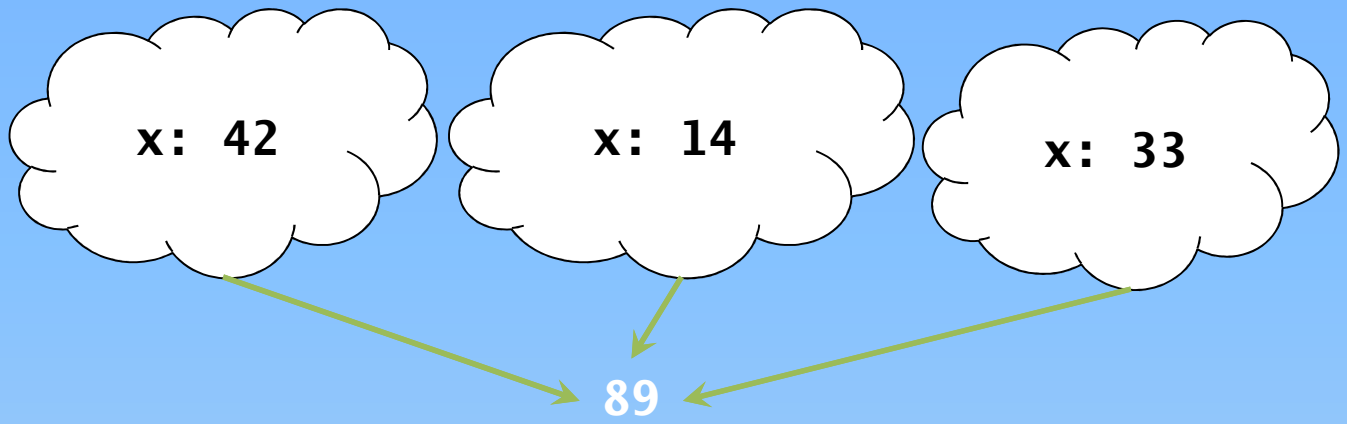
Изображение переменной – это её экземпляр. Потоки могут работать с переменной как с обычной нелокальной переменной.

При создании потока он получает собственное изображение переменной. В многопоточном приложении для переменной создаётся набор изображений.

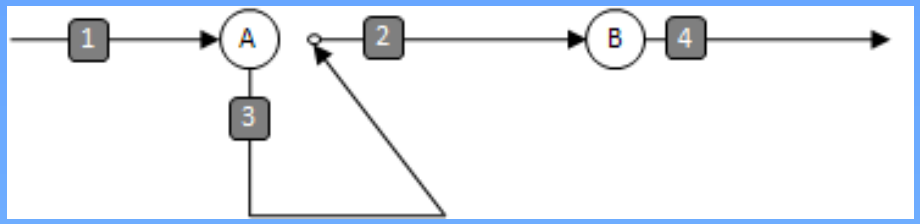
Система времени исполнения **Cilk Plus** координирует работу с изображениями переменной и объединяет их в точке объединения потоков (отсюда название - ***редукторы***).

Когда остаётся единственное изображение, оно устойчиво и значение переменной может быть извлечено из этого изображения.

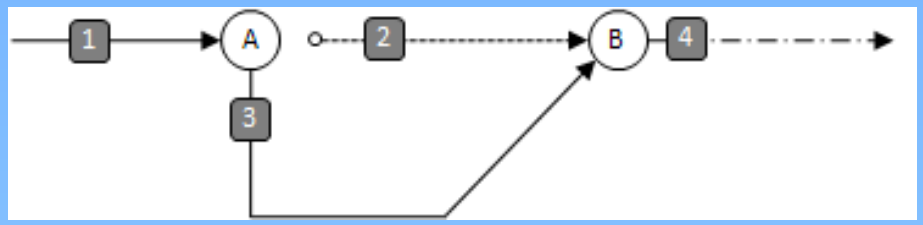
Редуктор суммирования:



Если в процессе выполнения Cilk-программы не происходит захвата работы, редуктор ведёт себя как обычная переменная.



Если происходит захват работы, потомок и продолжение получают собственные изображения.



Такую семантику иногда называют «ленивой».

Пример

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

class CilkForSum : public Sum {
public:
    virtual double FindSum(SimpleArray &data) {
        cilk::reducer_opadd<double> result(0);
        cilk_for(int i=0; i<data.GetSize(); i++)
            result += operation(data[i]);
        return result.get_value();
    }
};
```

Для того, чтобы использовать редуктор:

1. Добавить соответствующий заголовок.
2. Объявить переменную-редуктор как `reducer_kind<TYPE>`
3. Распараллелить цикл.
4. Получить результирующее значение с помощью метода `get_value()` после завершения цикла.

Пример

```
#include <iostream>
unsigned int compute(unsigned int i)
{
return i; // return a value computed from i
}
int main(int argc, char* argv[])
{
unsigned int n = 1000000;
unsigned int total = 0;
for(unsigned int i = 1; i <= n; ++i)
{
total += compute(i);
}
unsigned int correct = (n * (n+1)) / 2;

if (total == correct)
std::cout << "Total (" << total
<< ") is correct" << std::endl;
else
std::cout << "Total (" << total
<< ") is WRONG, should be "
<< correct << std::endl;
return 0;
}
```

```
#include <cilk.h>
#include <reducer_opadd.h>
#include <iostream>
unsigned int compute(unsigned int i)
{
return i;
}
int cilk_main(int argc, char* argv[])
{
unsigned int n = 1000000;
cilk::reducer_opadd<unsigned int> total;
cilk_for(unsigned int i = 1; i <= n; ++i)
{
total += compute(i); // Гонка за данными
}
unsigned int correct = (n * (n+1)) / 2;
if (total.get_value() == correct)
std::cout << "Total (" << total.get_value()
<< ") is correct" << std::endl;
else
std::cout << "Total (" << total.get_value()
<< ") is WRONG, should be "
<< correct << std::endl;
return 0;
}
```

Предопределённые редукторы

Редуктор/заголовок	Инициализация	Описание
`reducer_list_append <cilk/reducer_list.h>`	Пустой список	Объединение списков добавлением в конец
`reducer_list_prepend <cilk/reducer_list.h>`	Пустой список	Объединение списков добавлением в начало
`reducer_max <cilk/reducer_max.h>`	Аргумент конструктора	Нахождение максимального значения
`reducer_max_index <cilk/reducer_max.h>`	Аргумент конструктора	Нахождение максимального значения и его индекса в массиве
`reducer_min <cilk/reducer_min.h>`	Аргумент конструктора	Нахождение минимального значения
`reducer_min_index <cilk/reducer_min.h>`	Аргумент конструктора	Нахождение минимального значения и его индекса в массиве
`reducer_opadd <cilk/reducer_opadd.h>`	0	Суммирование

Редуктор/заголовок	Инициализация	Описание
`reducer_opand <cilk/reducer_opand.h>`	1/true	Логическое И
`reducer_opor <cilk/reducer_opor.h>`	0/false	Логическое ИЛИ
`reducer_opxor <cilk/reducer_opxor.h>`	0/false	Логическое исключающее ИЛИ
`reducer_ostream <cilk/reducer_ostream.h>`	Аргумент конструктора	Параллельный поток вывода
`reducer_basic_string <cilk/reducer_string.h>`	Пустая строка	Создание строки с помощью конкатенации
`reducer_string <cilk/reducer_string.h>`	Пустая строка	Создание строки с помощью конкатенации
`reducer_wstring <cilk/reducer_string.h>`	Пустая строка	Создание строки с помощью конкатенации

Список заголовочных файлов

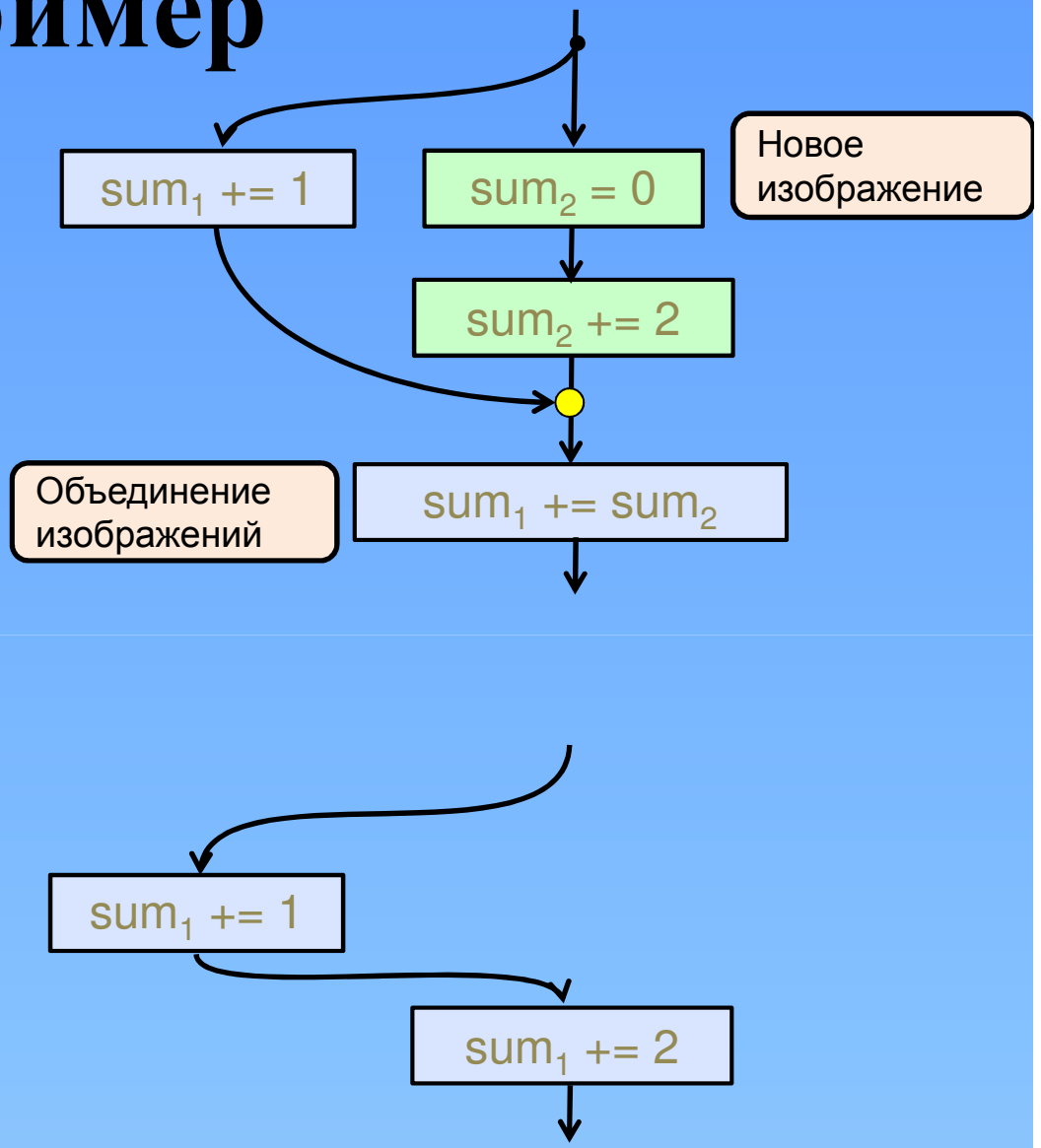
```
reducer.h
reducer_list.h
reducer_max.h
reducer_min.h
reducer_opadd.h
reducer_opand.h
reducer_opor.h
reducer_opxor.h
reducer_ostream.h
reducer_string.h
```

Пример

Глобальная переменная

```
 cilk::reducer_opadd<float> sum;

void f( int m ) {
    sum += m;
}

float g() {
    cilk_spawn f(1);
    f(2);
    cilk_sync;
    return sum.get_value();
}
```

Значение получаем с помощью get_value().



Еще один пример:

```
cilk::reducer_opadd<float> sum = 0;
...
cilk_for( size_t i=1; i<n; ++i )
    sum += f(i);
... = sum.get_value();
```

Пример

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

#include <math.h>

…

// CilkForSum class uses cilk_for keyword to compute the sum
in parallel
// cilk::reducer_opadd class is used for sync

class CilkForSum : public Sum {
public:
    virtual double FindSum(SimpleArray &data) {
        cilk::reducer_opadd<double> result(0);
        cilk_for(int i=0; i<data.GetSize(); i++)
            result += operation(data[i]);
        return result.get_value();
    }
};
```

Пользовательские гиперобъекты

Редуктор, определённый пользователем, состоит из 4-х логических частей:

1. **View** – класс, приватные данные редуктора. Конструктор должен инициализировать изображение.
2. **Monoid** – класс. Множество значений, ассоциативная операция на этом множестве и инициализирующее значение.
3. **Гиперобъект** – изображение для каждого потока.
4. **Остальная часть редуктора** – описывает, как выполняется доступ к данным и их модификация. `get_value()` возвращает значение.

Пример

```
#include <cilk/reducer.h>
class point
{
// Здесь – определение класса point
};
class point_holder
{
struct Monoid: cilk::monoid_base<point>
{
static void reduce (point *left, point *right) {}
};
private:
cilk::reducer<Monoid> imp_;
public:
point_holder() : imp_() {}
void set(int x, int y) {
point &p = imp_.view();
p.set(x, y);
}
bool is_valid() { return imp_.view().is_valid(); }
int x() { return imp_.view().x(); }
int y() { return imp_.view().y(); }
};
```

Cilkscreen

Cilk Plus позволяет выявлять гонки за данными независимо от числа потоков, при умеренных накладных расходах (памяти и процессорного времени).

Cilkscreen – инструмент выявления гонок за данными. Cilk-приложение выполняется один раз и Cilkscreen определяет наличие и положение гонок за данными.

! Большие накладные расходы: объём используемой памяти может увеличиваться в 4-5 раз, а время исполнения в 10-50 раз.

Для исследования Cilk-программы не требуется специальная подготовка (можно использовать Release-код).

Возможна привязка к исходному коду.

Определяются положение первого обращения к переменной и выполняется трассировка стека для второго доступа.

Выявлять гонки за данными независимо от числа потоков, при умеренных накладных расходах (памяти и процессорного времени).

Распознает блокировки.

Cilkscreen использует технологию Pin.

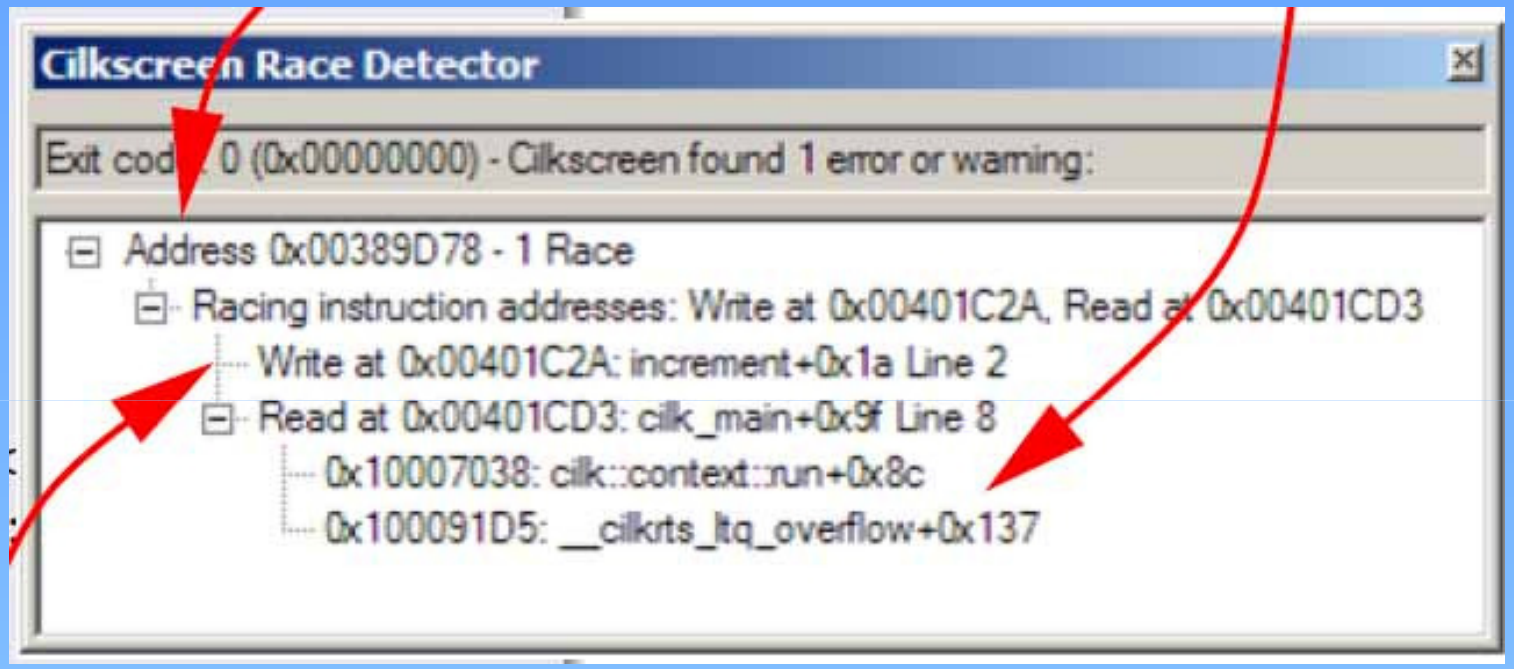
Cilkscreen распознаёт гонки за данными разного типа, но точность распознавания разная.

Пример

```
void increment(int& i)
{
  ++i;
}

int main()
{
 int x = 0;
      cilk_spawn increment(x);
 int y = x - 1;
return y;
}
```

Адрес гонки за данными

Трассировка стека для второго доступа



Первый доступ к переменной

Cilkview

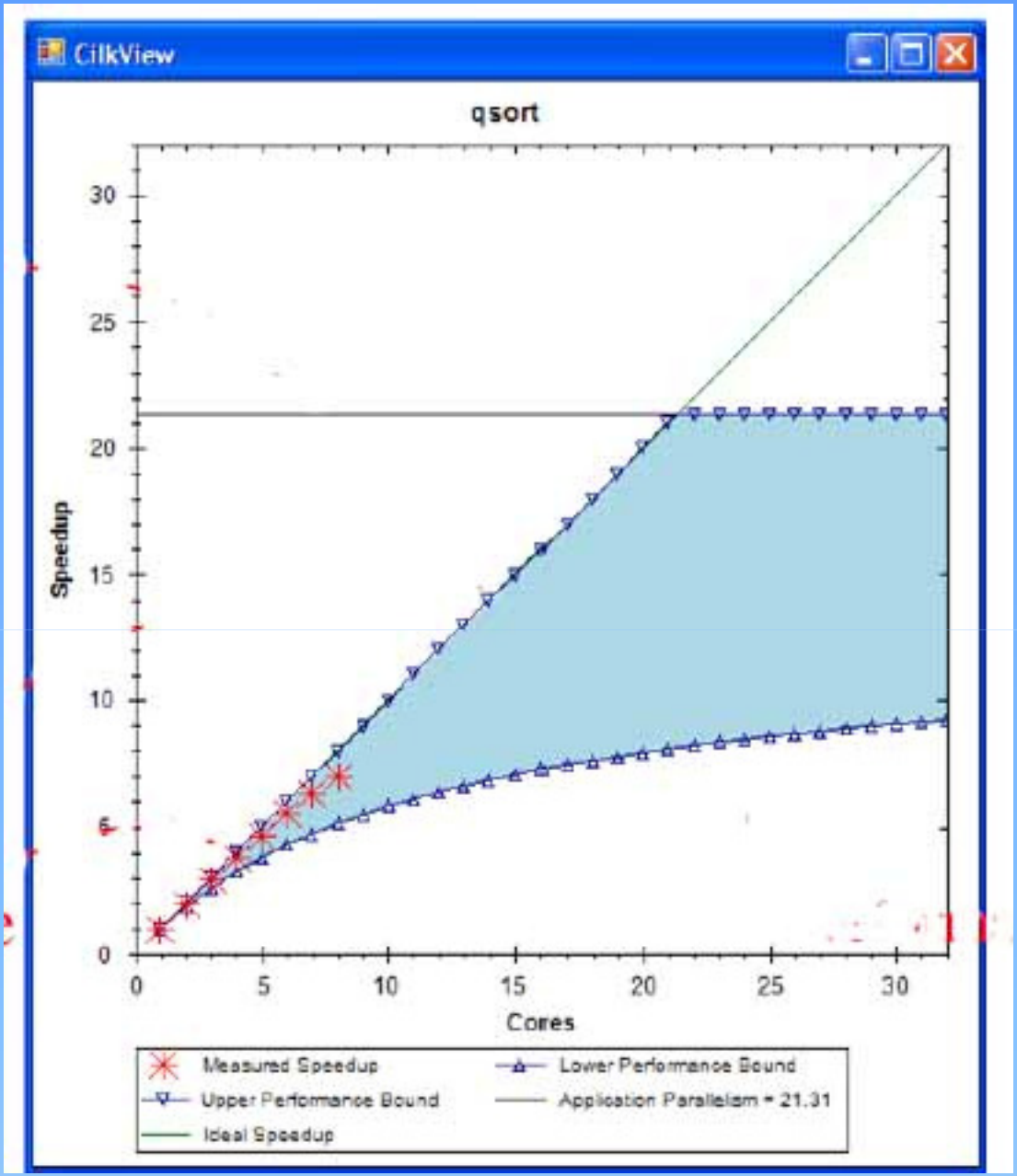
Cilkview позволяет анализировать масштабируемость (эффективность реализации параллелизма) Cilk-программы.

Специальная подготовка файла не требуется – можно использовать Release-версию исполняемого файла. Cilkview использует технологию Pin для инструментовки бинарного файла.

Подсчёт ведётся в инструкциях, а не единицах времени. Оцениваются работа и длина критического пути, нижняя и верхняя границы производительности.

Анализироваться могут фрагменты программы.

Небольшие накладные расходы: объём используемой памяти может практически не увеличиваться, а время исполнения возрастает не более, чем в 5 раз.


CilkView
qsort
Speedup
Cores
0
5
10
15
20
25
30
Measured Speedup
Upper Performance Bound
Ideal Speedup
Lower Performance Bound
Application Parallelism = 21.31

Расширенная индексная нотация

Язык программирования должен предоставлять разработчику удобное средство отображения параллелизма данных в задаче на параллельную архитектуру.

Языком, располагающим удобными и разнообразными средствами работы с массивами, является Fortran.

В C/C++ нет удобных средств работы с массивами. C/C++ - доминирующий язык разработки приложений.

Массивы – основная структура данных в вычислительных приложениях.

Расширенная индексная нотация – главное отличие **Cilk™** от **Cilk™ Plus**.

Определение:

```
<имя массива или указатель на него>[<нижняя граница значений индекса>:<длина>[: <шаг изменения индекса>]]
```

Символ : является указанием на множество элементов массива (*секцию* или *сечение* массива).

Символ «:», используемый без указания длины и шага, является указанием на множество всех элементов массива.

Синтаксис расширенной индексной нотации отличается от синтаксиса сечений в Fortran!

Использование расширенной индексной нотации является сигналом компилятору выполнить векторизацию кода.

Компилятор векторизует код с расширенной векторной нотацией, отображая его на целевую архитектуру

Примеры

```
A[:]          // Все элементы вектора A
B[3:5]        // Элементы с 3 по 5 массива B
C[:][7]       // Столбец 7 матрицы C
D[0:3:2]      // Элементы 0,2 и 4 массива D
E[0:5][0:4]   // 20 элементов с E[0][0] по E[5][4]
```

Большинство «стандартных» арифметических и логических операций C/C++ могут применяться к секциям массивов:

```
+, -, *, /, %, <,==,!=,>,|,&,^,&&,||,!,-(unary),
+(unary),++,--, +=, -=, *=, /=, *(p)
```

Операторы применяются ко всем элементам секции массива:

```
a[:] * b[:]                    // поэлементное умножение
a[3:2][3:2] + b[5:2][5:2]      // сложение матриц 2x2
```

Операции могут выполняться с разными элементами параллельно.

Секции, используемые в качестве операндов, должны быть конформными (иметь одинаковые ранг и экстент):

```
a[0:4][1:2] + b[1:2]  // так не должно быть!
```

Скалярный операнд автоматически расширяется до секции необходимой формы:

```
a[:][:] + b[0][1]    // сложение b[0][1] со всеми
                     // элементами матрицы a
```

Оператор присваивания выполняется параллельно для всех элементов секции:

```
a[0:n] = b[0:n] + 1;
```

Ранги правой и левой частей должны совпадать. Допустимо использование скалярных величин:

```
a[:] = c;          // c заполняет массив a
e[:] = b[:][:];    // ошибка!
```

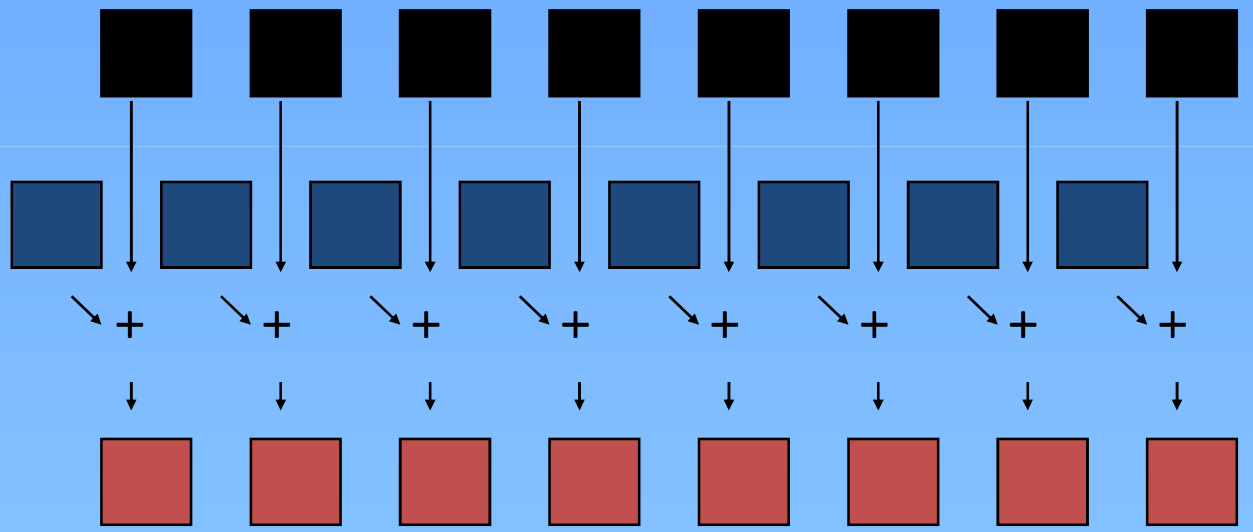
Допустимо пересечение правой и левой частей в операторе присваивания (в этом случае используются временные массивы):

```
a[1:s] = a[0:s] + 1; // используется старое значение
a[1:s-1]
```

Поэлементные векторные операции

Пример:

```
a[:]+b[:]
```



Пример. Сложение двух массивов

```
#include <iostream>
int main() {
   double a[4] = {1.,2.,3.,4.};
   double b[4] = {5.,7.,11.,13.};
   double c[4] = {0.,0.,0.,0.};

   std::cout << "Вывод a:\n" << a[:] << " ";
   std::cout << std::endl << std::endl;
   std::cout << "Вывод b:\n" << b[:] << " ";
   std::cout << std::endl << std::endl;

   std::cout << "Вывод c:\n" << c[:] << " ";
   std::cout << std::endl << std::endl;

   c[:] = a[:] + b[:];

   std::cout << "c = a + b:\n" << c[:] << " ";
   std::cout << std::endl << std::endl;
}
```

Пример. Операции с маской

```
#include <iostream>
int main() {
   bool x[4] = {0, 0, 1, 1};
   bool y[4] = {0, 1, 1, 0};
   double a[4] = {1.,2.,3.,4.};
   double b[4] = {5.,7.,11.,13.};
   double c[4] = {0.,0.,0.,0.};
   std::cout << "Вывод a:\n" << a[:] << " ";
   std::cout << std::endl << std::endl;

   std::cout << "Вывод b:\n" << b[:] << " ";
   std::cout << std::endl << std::endl;

   std::cout << "Вывод до c:\n" << c[:] << " ";
   std::cout << std::endl << std::endl;

   c[:] = x[:] && y[:] ? a[:] : b[:];

   std::cout << " Вывод после c:\n" << c[:] << " ";
   std::cout << std::endl << std::endl;
}
```

Пример. Реализация со встроенными функциями

```
#include <pmmintrin.h>
void foo(float* dest, short* src, long len, float a) {
   __m128 xmmMul = _mm_set1_ps(a);
   for(long i = 0; i < len; i+=8) {
      __m128i xmmSrc1i = _mm_loadl_epi64((__m128i*) &src[i]);
      __m128i xmmSrc2i = _mm_loadl_epi64((__m128i*) &src[i+4]);

      xmmSrc1i = _mm_cvtepi16_epi32(xmmSrc1i);
      xmmSrc2i = _mm_cvtepi16_epi32(xmmSrc2i);

      __m128 xmmSrc1f = _mm_cvtepi32_ps(xmmSrc1i);
      __m128 xmmSrc2f = _mm_cvtepi32_ps(xmmSrc2i);

      xmmSrc1f = _mm_mul_ps(xmmSrc1f, xmmMul);
      xmmSrc2f = _mm_mul_ps(xmmSrc2f, xmmMul);

      _mm_store_ps(&dest[i], xmmSrc1f);
      _mm_store_ps(&dest[i+4], xmmSrc2f);
   }
}
```

Сравнение машинных кодов для обеих реализаций

Интринсики (встроенные функции)

```
        movq      (%rsi,%rax,2), %xmm1                          #9.18
        movq      8(%rsi,%rax,2), %xmm2                         #10.18
        pmovsxwd  %xmm1, %xmm1                                  #9.18
        pmovsxwd  %xmm2, %xmm2                                  #10.18
        cvtdq2ps  %xmm1, %xmm3                                  #12.25
        cvtdq2ps  %xmm2, %xmm4                                  #13.25
        mulps     %xmm0, %xmm3                                  #15.18
        mulps     %xmm0, %xmm4                                  #16.18
        movaps    %xmm3, (%rdi,%rax,4)                          #18.21
        movaps    %xmm4, 16(%rdi,%rax,4)     ,                  #19.21
        addq      $8, %rax                                      #5.34
        cmpq      %rdx, %rax                                    #5.24
        jl        ..B1.3        # Prob 82%                      #5.24
```

Сравнение машинных кодов для обеих реализаций

Индексная нотация

```
movq      (%rsi,%rcx,2), %xmm2                    #2.45
pmovsxwd  %xmm2, %xmm2                            #2.45
cvtdq2ps  %xmm2, %xmm3                            #2.45
mulps     %xmm1, %xmm3                            #2.58
movaps    %xmm3, (%rdi,%rcx,4)                    #2.15
movq      8(%rsi,%rcx,2), %xmm4                   #2.45
pmovsxwd  %xmm4, %xmm4                            #2.45
cvtdq2ps  %xmm4, %xmm5                            #2.45
mulps     %xmm1, %xmm5                            #2.58
movaps    %xmm5, 16(%rdi,%rcx,4)  ,               #2.15
addq      $8, %rcx                                #2.15
cmpq      %r8, %rcx                               #2.15
jb        ..B1.15       # Prob 44%                #2.15
```

Операции сбора/распределения данных

В качестве индекса массива можно использовать сечение массива. Элементы сечения в этом случае определят множество значений индекса.

Примеры:

```
a[b[0:s]] = c[:] // a[b[0]]=c[0],a[b[1]]=c[1],…

c[0:s] = a[b[:]] // c[0]=a[b[0]],c[1]=a[b[1]],…
```

Компилятор при использовании таких конструкций генерирует машинный код операций сбора и распределения данных для соответствующей целевой архитектуры.

Пример использования операций распределения/сбора данных

```
void fnc1(float *dest, float *src, unsigned int *ind_dest, unsigned int
*ind_src, int len) {
   dest[ind_dest[0:len]] = src[ind_src[0:len]];
}

#include <iostream>
void foo(float *dest, float *src, unsigned int *ind_dest, unsigned int
*ind_src, int len);

int main() {
   float x[5] = {1., 2., 3., 4., 5.};
   float y[5] = {0., 0., 0., 0., 0.};
   unsigned int y_ind[5] = {4,3,2,1,0};
   unsigned int x_ind[5] = {1, 3, 0, 2, 4};
 std::cout << "x: " << x[:] << " ";
   std::cout << std::endl << std::endl;
   std::cout << "y: " << y[:] << " ";
   std::cout << std::endl << std::endl;
   fnc1(y, x, y_ind, x_ind, 5);
   std::cout << "y: " << y[:] << " ";
   std::cout << std::endl << std::endl;
   return(0);
}
```

Операции линейного/циклического сдвига

Поддерживаются операции линейного и циклического («ротация») сдвига.

Примеры:

```
b[:] = __sec_shift(a[:], shift_val, fill_value);
b[:] = __sec_rotate(a[:], shift_val);
```

Параметр `shift_val` определяет величину сдвига, а `fill_value` - значение, которым заполняются «освободившиеся» позиции массива `a`.

О многомерных массивах

Для работы с сечениями многомерных массивов компилятору необходимо «знать» форму массива.

В C/C++ есть следующие способы описания формы массива:

массив фиксированного размера:

```
float a[100][50];
```

динамический массив:

```
typedef int (*a2d)[10]; // указатель на вектор
a2d *p;
p = (a2d) malloc(sizeof(int)*rows*10);
p[5][:] = 42;          // задать элементы 5-й строки
p[0:rows][:] = 42; // задать все элементы массива
p[:][:] = 42;          // ошибка (размер строки должен
                       // быть задан явно)
```

Массивы как аргументы

Сечение массива можно использовать в качестве аргумента функции.

Фактические и формальные аргументы должны быть согласованы.

Пример:

```
void saxpy_vec(int m, float a, float restrict x[m],
float restrict y[m])
{
    y[:] += a * x[:];
}

cilk_for(int i = 0; i < n; i += 256)
    saxpy_vec(112, 1.7, &x[i],&y[i]);
```

Примеры

Модификация подматрицы размером `m x n`, начиная с элемента `(i, j)`:

```
vx[i:m][j:n] += a*(U[i:m][j+1:n]-U[i:m][j:n]);
```

Использование элементной функции:

```
theta[0:n] = atan2(y[0:n],1.0);
```

Сбор/распределение данных:

```
w[0:n] = x[i[0:n]];
y[i[0:n]] = z[0:n];
```

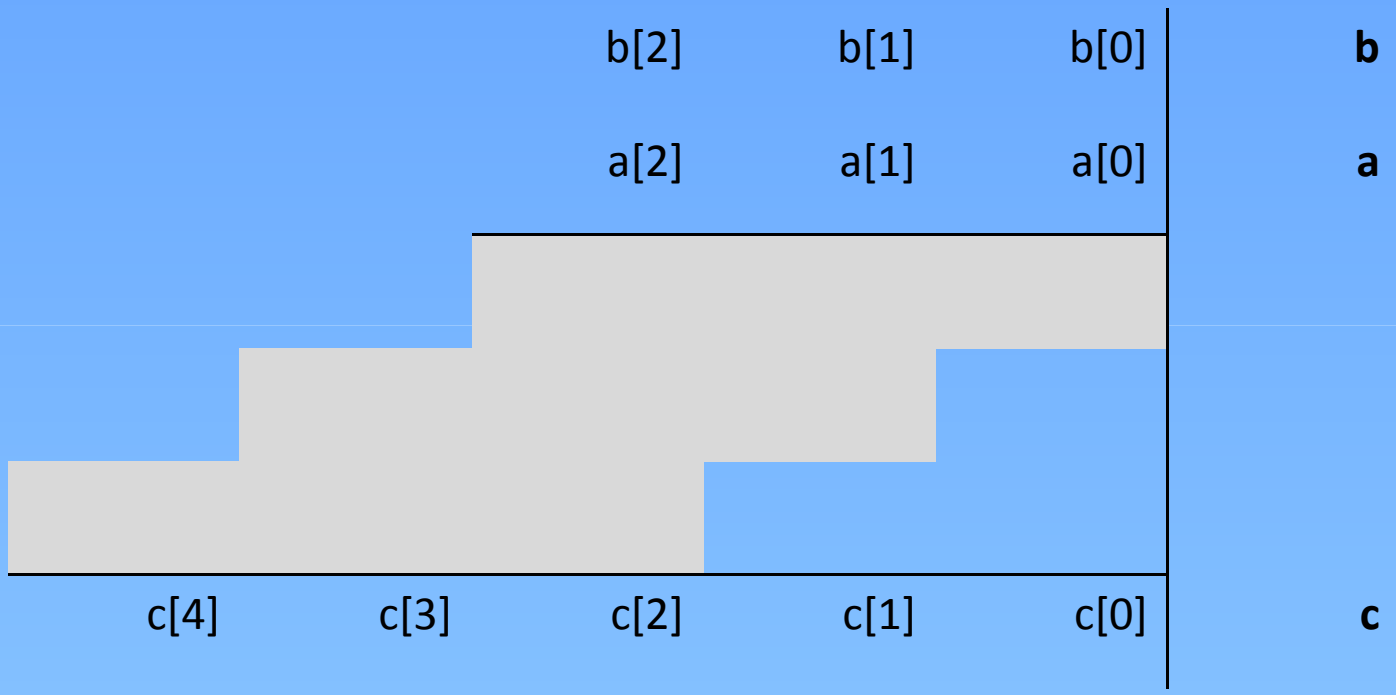
Использование сечения массива в условном операторе (выполняются обе ветви):

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

Пример. Умножение полиномов c = a·b

			x^2	+	$2x$	+	3	**b**
			x^2	+	$4x$	+	5	**a**
			$5x^2$	+	$10x$	+	15	
		$4x^3$	+	$8x^2$	+	$12x$		
$4x^4$	+	$2x^3$	+	$3x^2$				
$4x^4$	+	$6x^3$	+	$16x^2$	+	$22x$	+ 15	**c**

Хранение коэффициентов



Векторная реализация (сложность $\Theta(n^2)$)

```
void simple_mul( T c[], const T a[], const T b[],
size_t n ) {
    c[0:2*n-1] = 0;
    for (size_t i=0; i<n; ++i)
        c[i:n] += a[i]*b[0:n];
}
```

Алгоритм Карацубы (сложность $\Theta(n^{1.5})$) – оптимален для n = 32 – 1024

$K = x^{\lfloor n/2 \rfloor}$

$$a = a_1 K + a_0$$
$$b = b_1 K + b_0$$

Вычислить:

$$t_0 = a_0 \cdot b_0$$
$$t_1 = (a_0 + a_1) \cdot (b_0 + b_1)$$
$$t_2 = a_1 \cdot b_1$$

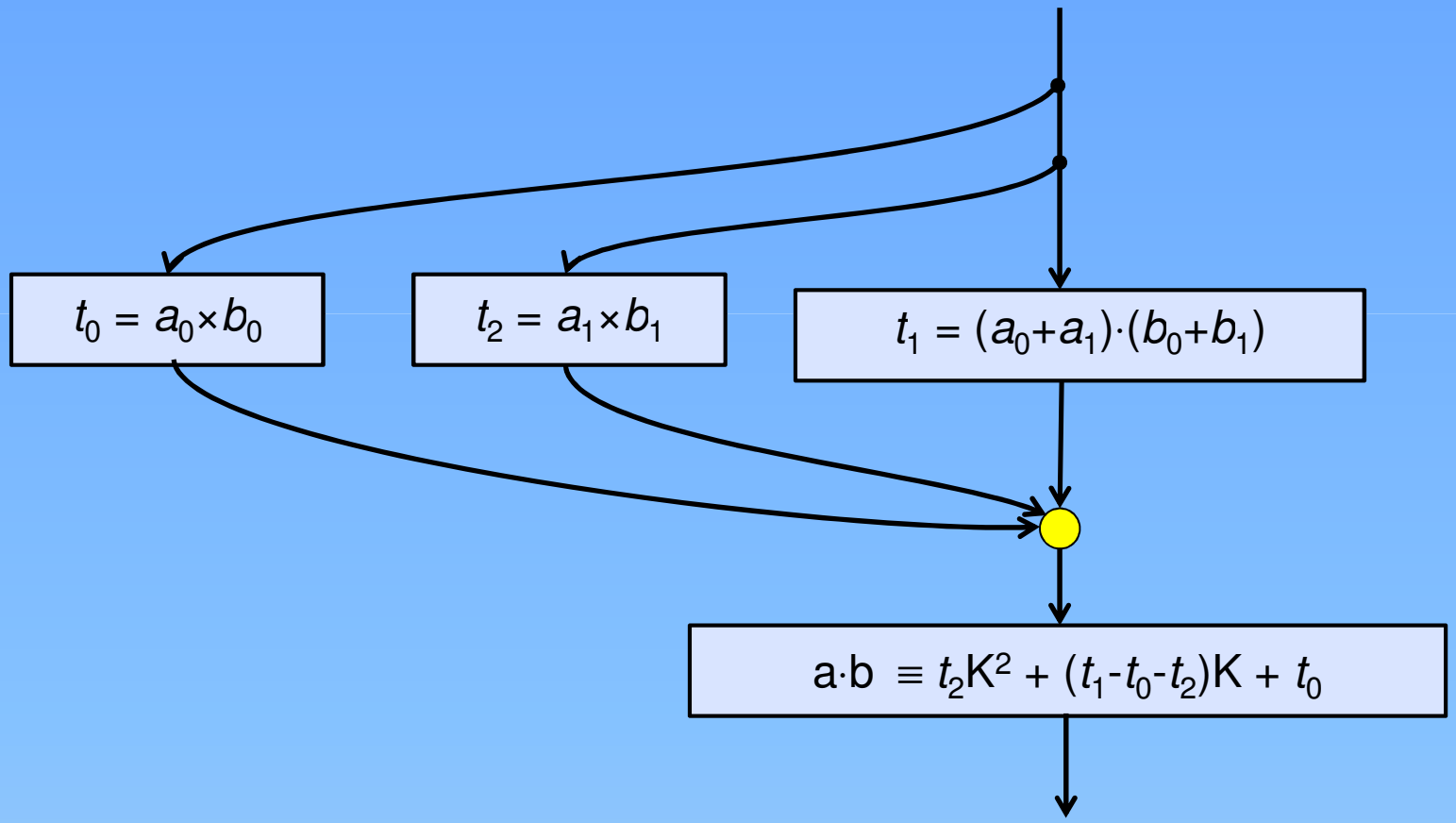
Тогда:

$$a \cdot b \equiv t_2 K^2 + (t_1 - t_0 - t_2) K + t_0$$

Реализация с помощью сечений

```
void karatsuba( T c[], const T a[], const T b[], size_t n ) {
 if( n<=CutOff ) {
     simple_mul( c, a, b, n );
 } else {
     size_t m = n/2;
     karatsuba( c, a, b, m );              // t0 = a0×b0
     karatsuba( c+2*m, a+m, b+m, n-m ); // t2 = a1×b1
     temp_space<T> s(4*(n-m));
     T *a_=s.data(), *b_=a_+(n-m), *t=b_+(n-m);
     a_[0:m] = a[0:m]+a[m:m];              // a_ = (a0+a1)
     b_[0:m] = b[0:m]+b[m:m];              // b_ = (b0+b1)
     karatsuba( t, a_, b_, n-m );          // t1 = (a0+a1)×(b0+b1)
     t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1];// t = t1-t0-t2
     c[2*m-1] = 0;
     c[m:2*m-1] += t[0:2*m-1];    // c = t2K^2+(t1-t0-t2)K+t0
  }
}
```

Схема распараллеливания



Эффективная векторизация – как её добиться?

Векторизация в рамках C/C++. Недостаток – не всегда удобная реализация.

Использование расширения процессорных инструкций SSE (Streaming SIMD Extension). Недостаток – фиксированная длина вектора.

Использование элементных операций/функций Intel® Cilk™ Plus.

Элементные функции

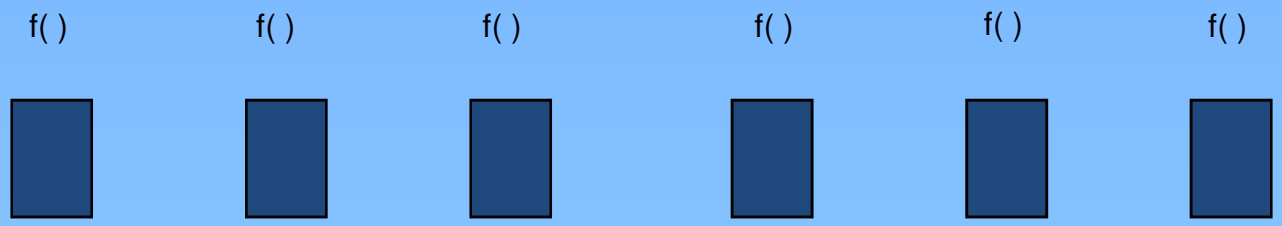
Элементные функции формируют результат вычисления скалярной функции для каждого элемента массива (вызов скалярной функции с векторным аргументом формирует массив значений, конформный аргументу):

```
__declspec(vector) <сигнатура функции>
```

Отображение заменяет цикл последовательной программы.

Примеры:

```
a[:] = sin(b[:]);
a[:] = pow(b[:], c); // b[:]**c
a[:] = pow(c, b[:]); // c**b[:]

f(b[:])
```



При компиляции кода с вызовом элементных функций компилятор генерирует обращения к векторизованным функциям.

Компилятор может генерировать многопоточный код.

Если функция определена как `elemental`, компилятор генерирует векторизованный код для этой функции.

Исключены побочные эффекты.

Функции отображаются параллельно!

Ограничения

1. Допускается использование только следующих типов:
   - signed/unsigned 8/16/32/64 битовые целые;
   - 32 или 64 битовые с плавающей точкой;
   - 64 или 128 битовые комплексные;
   - указатель или ссылка C++.

2. Не допускается использование ключевых слов `for`, `while`, `do`, `goto`.
3. Не допускается использование операторов выбора.
4. Не допускается использование ассемблерных вставок.
5. В функциях не допускается многопоточность, реализованная с помощью , OpenMP, cilk_spawn/cilk_for.
6. Не допускаются виртуальные функции и указатели на функции.
7. В функциях не допускается использование выражений с индексной нотацией.

и другие.

Пример

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


void saxpy(float a, float *x, float *y) {
   *y += a * (*x);
}
```

```
__declspec(vector(scalar(a),linear(x),linear(y)))
void saxpy(float a, float *x, float *y);
void foo(float *restrict x, float *restrict y, float a, int
len) {
      saxpy(a, x[0:len], y[0:len]);
}
```

Компилятор сгенерирует вот такой код:

```
void saxpy_4(float a, float x[4],  float y[4])
{
    y[:] += a * x[:];
}
for(i = 0; i < len-3; i += 4) {
    saxpy_4(a, &x[i], &y[i]);
}
for(; i < len; i++) {
    saxpy(a, &x[i], &y[i]);
}
```

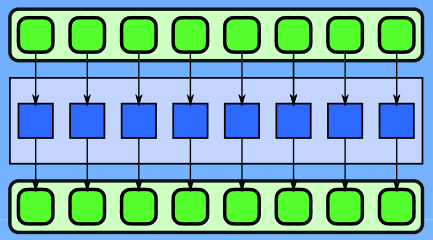
Примеры применения (по предметным областям):

- обработка изображений (гамма-коррекция и т.д.);
- преобразование между цветовыми пространствами;
- моделирование методом Монте-Карло.

Дополнительные примеры использования отображения функций

Параллелизм потоков

```
cilk_for( int i=0; i<n; ++i )
    a[i] = f(b[i]);
```



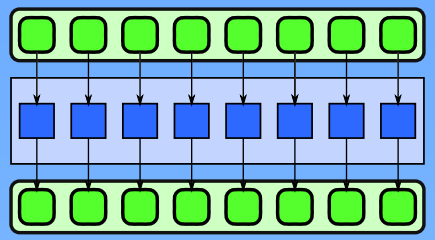
Векторный параллелизм

```
a[0:n] = f(b[i:n]);
```

```
#pragma simd
for( int i=0; i<n; ++i )
    a[i] = f(b[i]);
```

Пример использования отображения функций в Intel® Threading Building Blocks

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



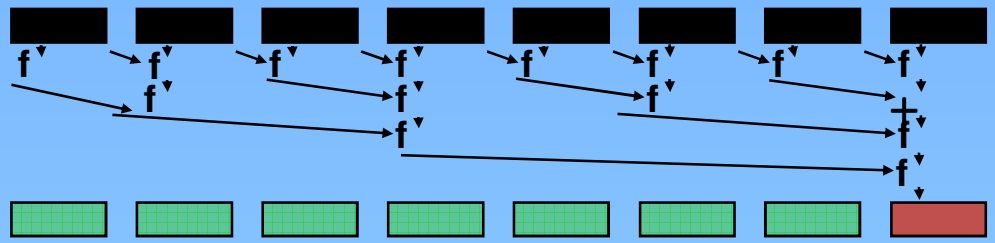
```
parallel_for(
    blocked_range<int>(0,n),
    [&](blocked_range<int> r ) {
        for( int i=r.begin(); i!=r.end(); ++i
)
            a[i] = f(b[i]);
    });
```

Операции приведения (редукции)

Операция редукции применяется к сечению массива. Её результат – скалярное значение.

Примеры:

```
__sec_reduce(f, a[:])
__sec_reduce_add(a[:])
```



Базовые типы C поддерживаются следующими операциями редукции:

```
add
mul
max
max_ind
min
min_ind
all_zero
all_non_zero
any_nonzero
```

Есть возможность определить пользовательскую операцию приведения:

```
type fn(type in1, type in2);
out = __sec_reduce(fn, identity_value, in[x:y:z]);
```

Пример использования пользовательской операции редукции

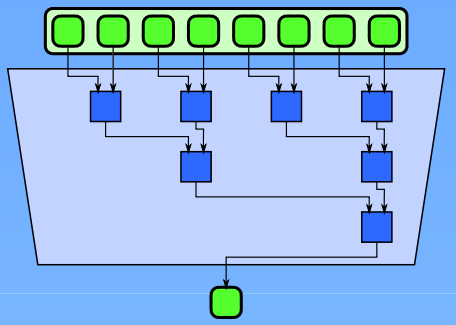
```
#include <iostream>

unsigned int bitwise_and(unsigned int x, unsigned int y) {
   return (x & y);
}

int main() {
   unsigned int a[4] = {5,7,13,15};
   unsigned int b = 0;

   std::cout << "Display a:\n" << a[:] << " ";
   std::cout << std::endl << std::endl;

   b = __sec_reduce(bitwise_and, 0xffffffff, a[:]);

   std::cout << "b:\n" << b << std::endl;
   return(0);
}
```

Примеры применения (по предметным областям):

- матричные операции;
- обработка изображений;
- вычисление средних при моделировании методом Монте-Карло.

Дополнительные примеры использования операции редукции

```
float sum = __sec_reduce_add(a[i:n]);
```

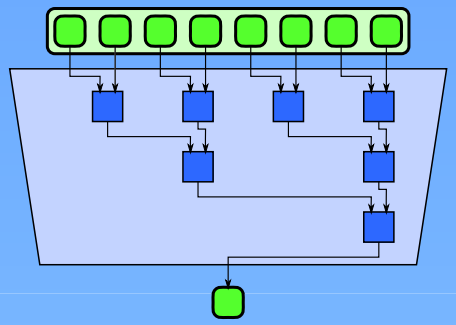


```
#pragma simd reduction(+:sum)
float sum=0;
for( int i=0; i<n; ++i )
    sum += a[i];
```

```
cilk::reducer_opadd<float> sum = 0;
cilk_for( int i=0; i<n; ++i )
    sum += a[i];
... = sum.get_value();
```

Пример использования операции редукции в Intel® Threading Building Blocks

```
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>()
);
```

Функции прикладного программного интерфейса

Функции ППИ позволяют управлять поведением программы.

Функции прикладного программного интерфейса (ППИ) используются с заголовочным файлом `cilk/cilk_api.h`

```
int __cilkrts_set_param(const char* name, const char* value);
```

Эта функция используется для управления некоторыми параметрами системы исполнения Cilk.
Первые 2 параметра строковые.
`nworkers` – значение определяет количество исполнителей. Если данная функция не используется, количество исполнителей задаётся с помощью переменной окружения `CILK_NWORKERS` или, по умолчанию, оно равно количеству ядер.

Данная функция действует только до первого использования `cilk_spawn` или `cilk_for`.

```
int __cilkrts_get_nworkers(void);
```

Эта функция возвращает количество потоков-исполнителей и фиксирует его так, что оно не может быть изменено вызовом функции `__cilkrts_set_param`.

пользуется для управления некоторыми параметрами системы исполнения Cilk.

Идентификаторы исполнителей не обязательно прнимают непрерывный (последовательный) ряд значений.

```
int __cilkrts_get_worker_number(void);
```

Эта функция возвращает целое значение, показывающее исполнителя, который выполняет функцию.

```
int __cilkrts_get_total_workers(void);
```

Эта функция возвращает суммарное количество потоков-исполнителей, включая неактивные.

Несколько советов по повышению производительности

Оптимизируйте в первую очередь последовательный код.

Выбор зернистости:

- избегайте порождения маленьких задач;
- оптимизируйте зернистость параллельных циклов;
- мелкозернистая декомпозиция => большие накладные расходы;
- крупнозернистая декомпозиция => низкий параллелизм, неэффективное использование возможностей вычислительной системы.

Оптимизируйте кэш-эффективность.

Пример false-sharing:

```
volatile int x[32];
void f(volatile int *p)
{
for (int i = 0; i < 100000000; i++)
{
++p[0];
++p[16];
}
}
int main()
{
cilk_spawn f(&x[0]);
cilk_spawn f(&x[1]);
cilk_spawn f(&x[2]);
cilk_spawn f(&x[3]);
cilk_sync;
return 0;
}
```