ИНСТРУМЕНТЫ INTEL ДЛЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

С.А.Немнюгин

Санкт-Петербургский Государственный Университет

s.nemnyugin@spbu.ru

nemnyugin@parserplus.com

Высокопроизводительные вычисления на GRID системах. Архангельск, 2014


HPC
Платформа
Параллельные технологии
Модель
Оптимизация готовой программы
Алгоритмы
Оптимизация при компиляции
Высокопроизводительные библиотеки
Программирование

Intel® Math Kernel Library

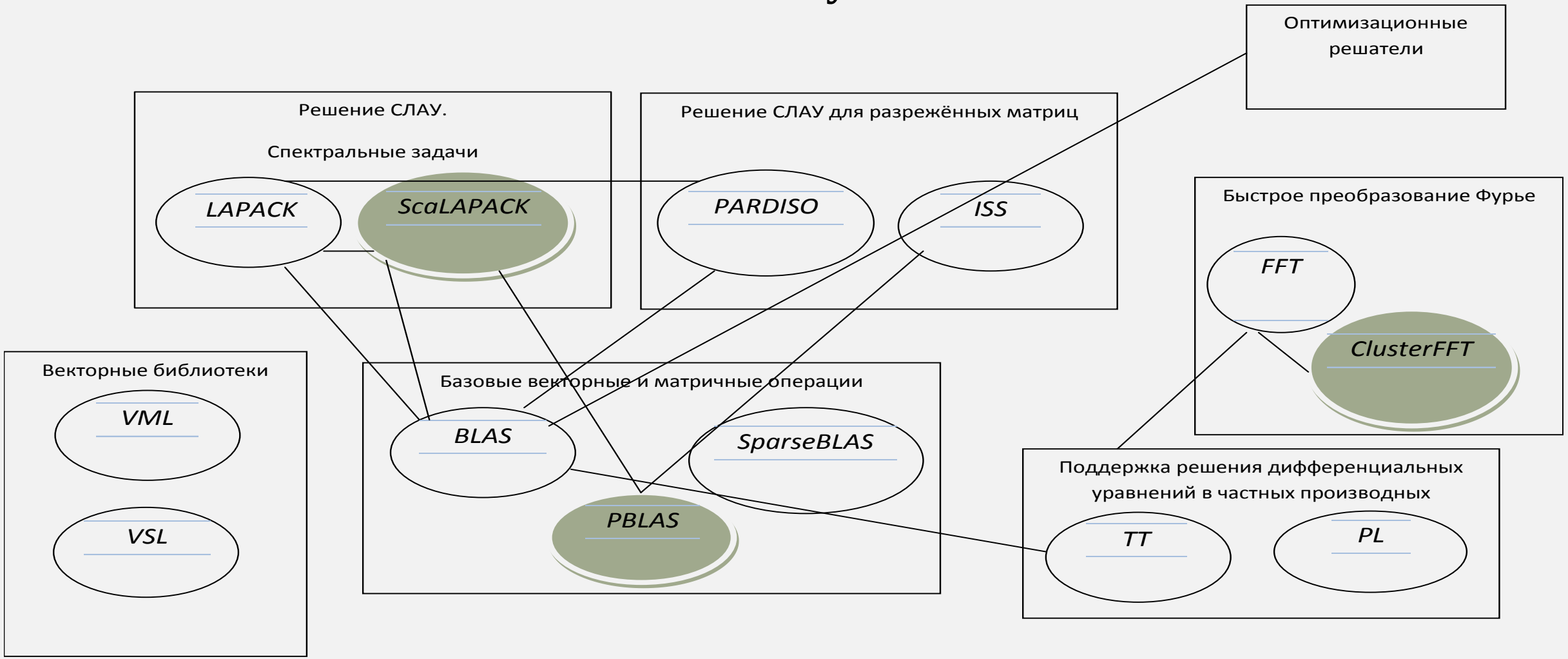
Intel® Math Kernel Library (Intel® MKL) – вычислительная математическая библиотека, содержащая хорошо оптимизированные многопоточные реализации математических функций. Содержит BLAS, LAPACK, ScaLAPACK, солверы для разреженных систем, быстрое преобразование Фурье, векторизованные математические функции и т.д.

Многоплатформенная библиотека, однако ScaLAPACK не поддерживается в Mac OS* X.

Поддерживаются C и Fortran.

Входит в состав следующих пакетов:
- IntelR Parallel Studio XE
- IntelR Cluster Studio XE
- IntelR C++ Studio XE
- IntelR Composer XE
- IntelR C++ Composer XE
- IntelR Fortran Composer XE

Intel® Math Kernel Library



Intel® Math Kernel Library (Intel® MKL) – оптимизированная библиотека.

Примеры методов оптимизации, использованных в MKL:

- Раскрутка циклов.
- Упаковка данных с целью улучшения их повторного использования.
- Применение приёмов работы с данными, которые предотвращают вытеснение данных из кэш-памяти.
- Предвыборка данных с целью преодоления латентности памяти.
- Векторизация и другие.

Общая характеристика и состав Intel® Integrated Performance Primitives

Библиотека готовых компонентов для разработки мультимедийных приложений для вычислительных платформ Intel.

Включает модули для обработки сигналов и выполнения векторных и матричных операций (**эффективна при работе с маленькими матрицами**), функции сжатия и распаковки речи и статических/динамических изображений, средства шифрования и обработки аудиоданных и текстовых строк.

Intel IPP обеспечивает прозрачное использование расширенных возможностей процессоров Intel, таких, как технология MMX, расширения набора команд Streaming SIMD Extensions и Streaming SIMD Extensions.

Библиотека Intel IPP оптимизирована для работы с разными процессорами Intel.

Библиотека Intel IPP поддерживает 32- и 64-битные операционные системы Windows и Linux, включая встраиваемые версии, такие как Windows Mobile.

Состав:

- Функции размещения и освобождения памяти.
- Вспомогательные функции – определение числа ядер, типа процессора, выравнивание указателей, определение числа потоков и т.д.
- Функции диспетчеризации.
- Функции инициализации векторов.
- Генерация сигналов разного типа.
- Генераторы однородного и неоднородных распределений.
- Логические, арифметические функции, функции преобразования векторов (сортировка, преобразование типов, объединение векторов, комплексное сопряжение и др.).
- Работа с окнами (обработка сигналов).
- Статистические функции.
- Функции фильтрации.
- Преобразования: FFT, Харли, Гильберта, Уолша-Адамара, вейвлетное.
- Функции распознавания и кодирования речи.
- Работа со строками.
- Элементарные и специальные математические функции.
- Сжатие данных и т.д.

Intel® Cilk™ Plus

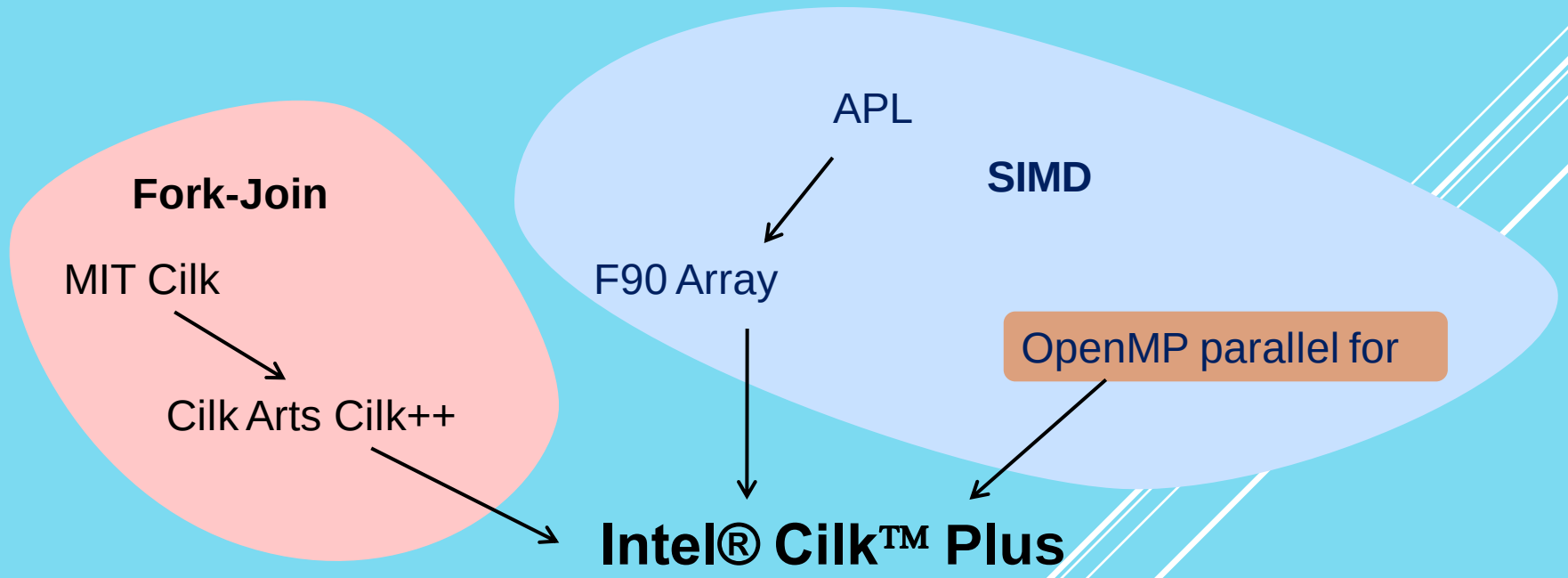
- расширение языков C/C++ (ключевые слова, расширенная векторная нотация, гиперобъекты, элементные функции);
- обеспечивает эффективный и безопасный параллелизм типа «fork-join» (операция порождения – spawn, гиперобъекты, диспетчеризация системой исполнения);
- обеспечивает векторный параллелизм (векторизация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

Intel® Cilk™ Plus поддерживается компиляторами:

- ✓ Intel (начиная с версии 12);
- ✓ GCC (начиная с версии 4.7.

Мультиплатформенность (Windows и Linux).

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


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);
    cilk_spawn sample_qsort(begin, middle);
    sample_qsort(++middle, ++end);
    cilk_sync;
,      }
}
```

Программа с использованием **Intel® Cilk™ Plus** 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 система исполнения. Балансировка выполняется методом *захвата работы*. Алгоритмы диспетчеризации таковы, что их эффективность, как правило, высока.

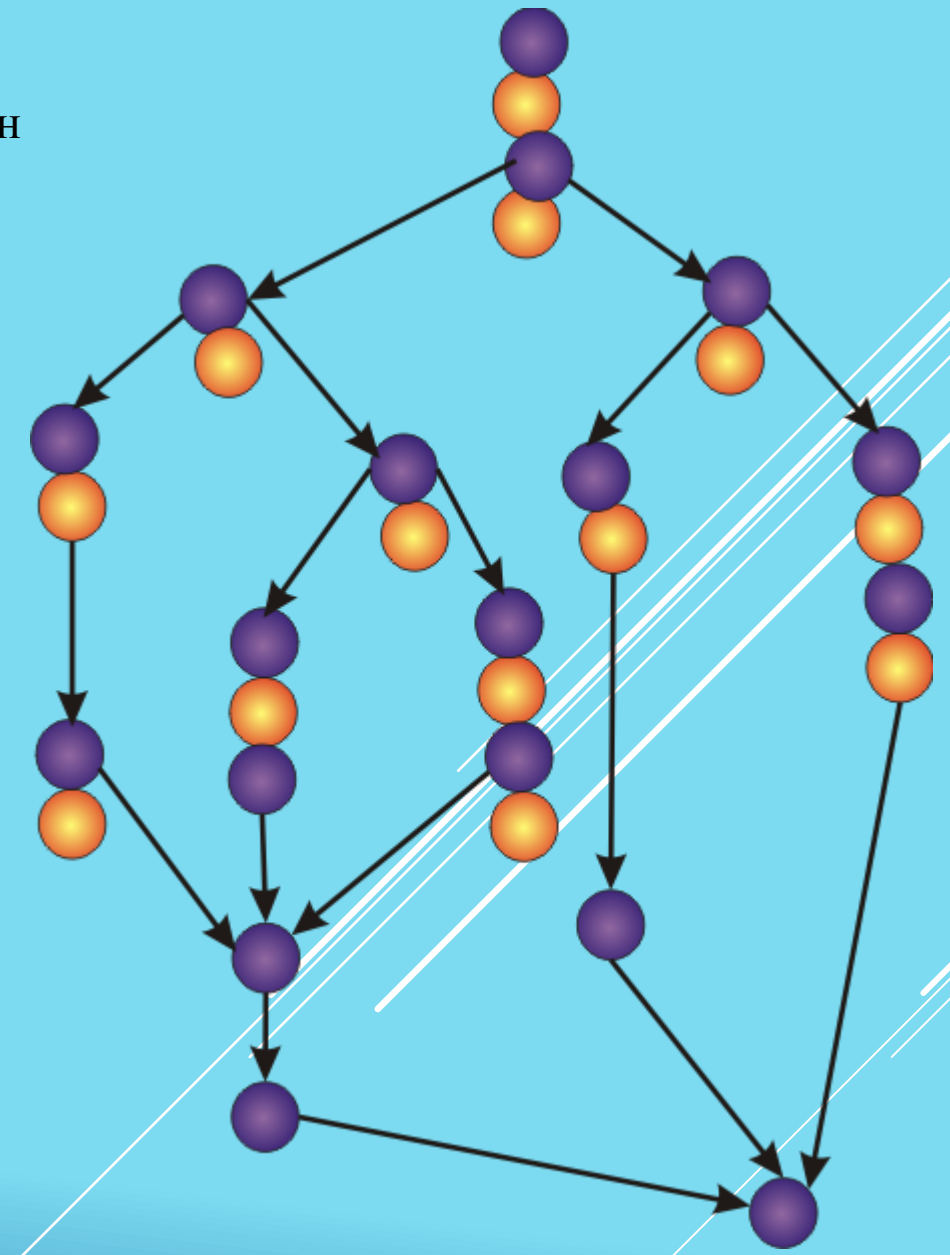
Программист
Определяет и описывает потенциальный параллелизм.

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.

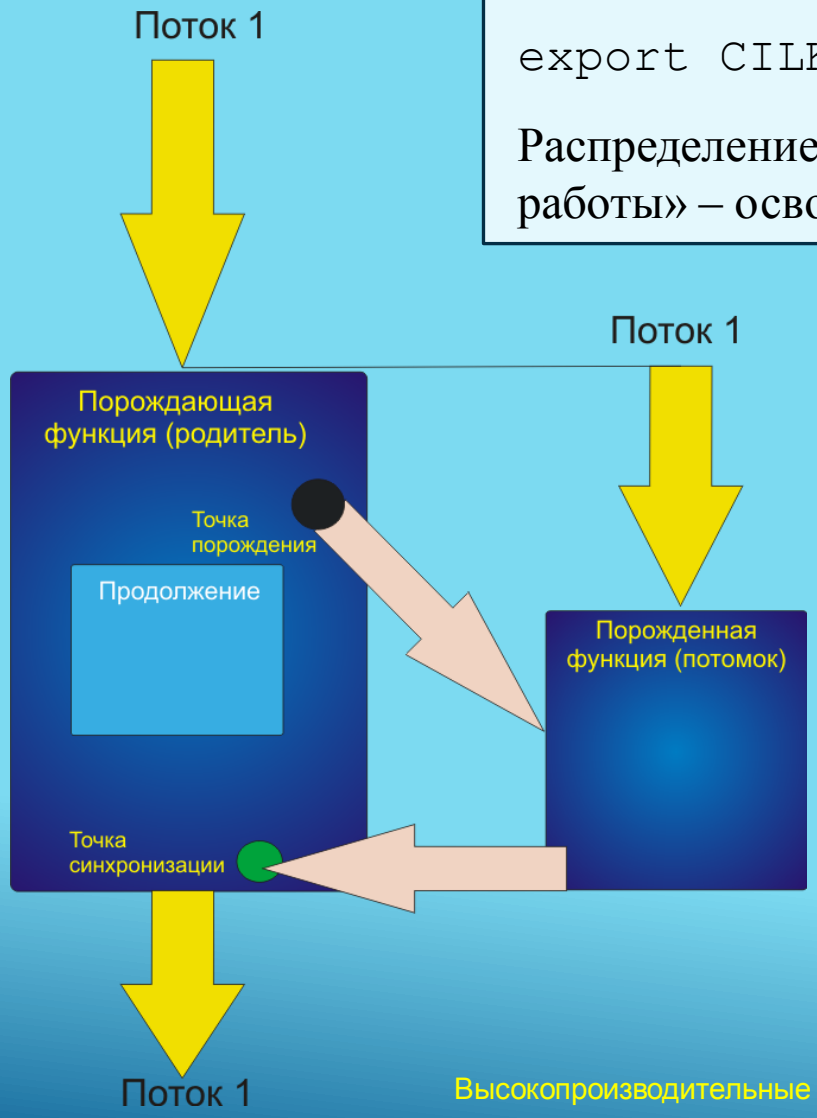
- «Ветви» - последовательные фрагменты кода. Исполняются в режимах «продолжения» и «порождения».
- Узлу порождения соответствуют 2 «наследника».



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

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

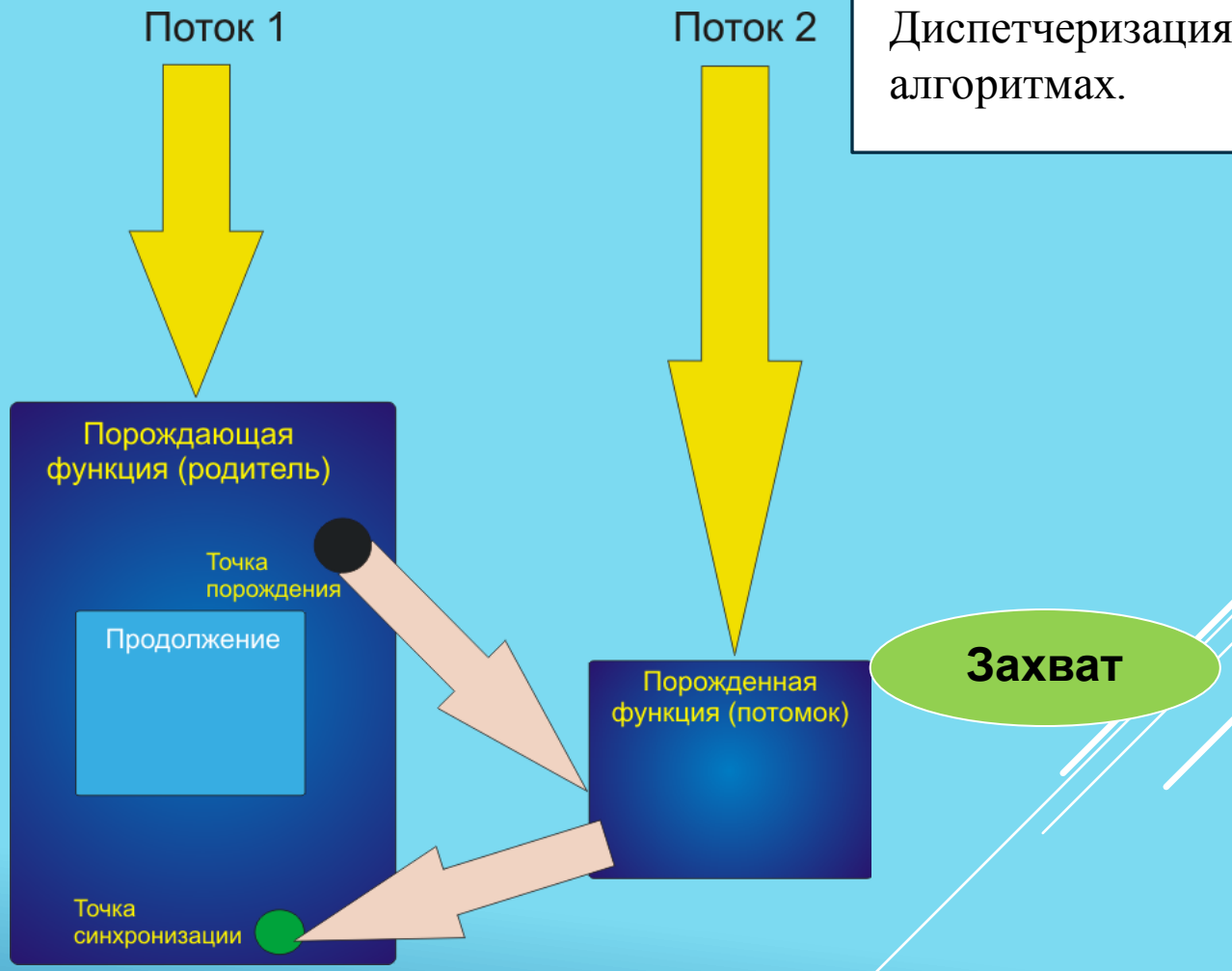
Распределение задач между потоками выполняется методом «захвата работы» – освободившийся поток выполняет очередную задачу.



Если доступен только один поток, программа выполняется как последовательная

Если доступно несколько потоков, программа выполняется как параллельная.

Диспетчеризация основана на «жадных» (greedy) алгоритмах.



Жадный алгоритм (англ. **Greedy algorithm**) — **алгоритм**, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

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.

Файл заголовков

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

Переменные окружения

Основная - `CILK_NWORKERS` (количество потоков).

Ключевые слова

Ключевое слово `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.

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`

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();
    }
```

Ключевое слово `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:

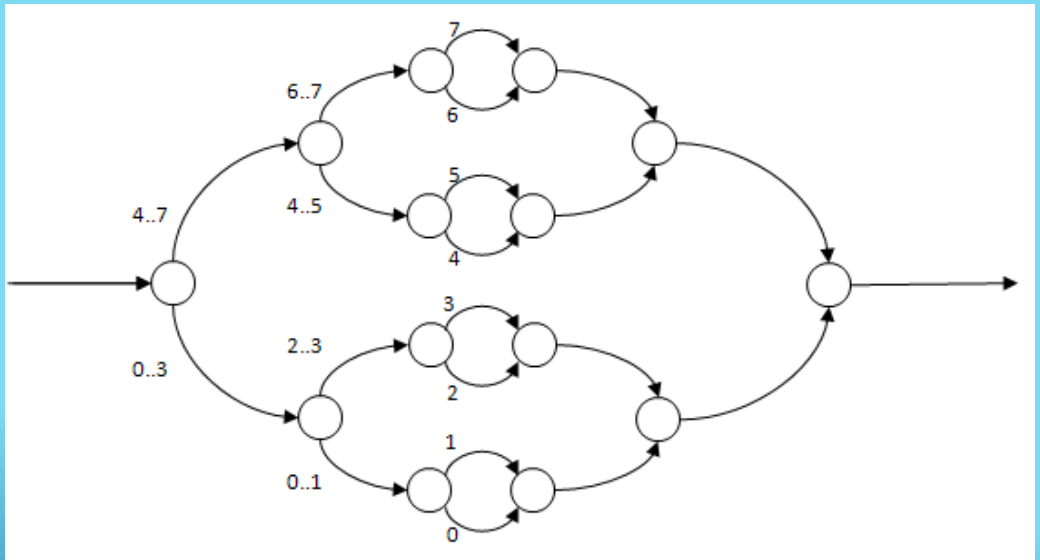
```
void floyd_warshall() {
      cilk_for (int k = 0; k < n; ++k)
            for (int i = 0; i < n; ++i)
                  work(k,i);
}
```

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).

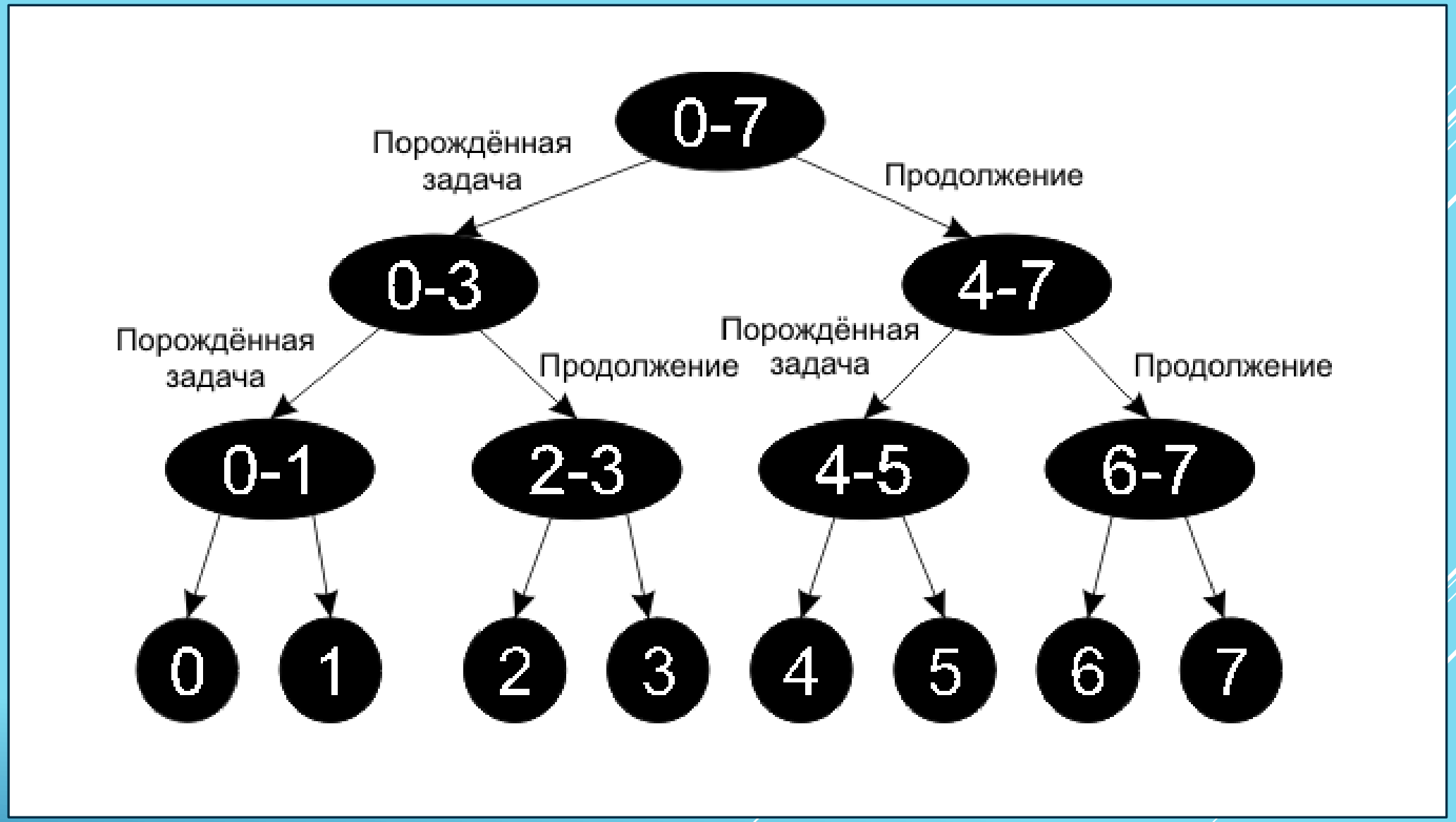
Если зернистость не указана явно, используется следующая формула:

```
#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 опытным путём!

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 А выше, чем эффективность распараллеливания фрагмента В.

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*.

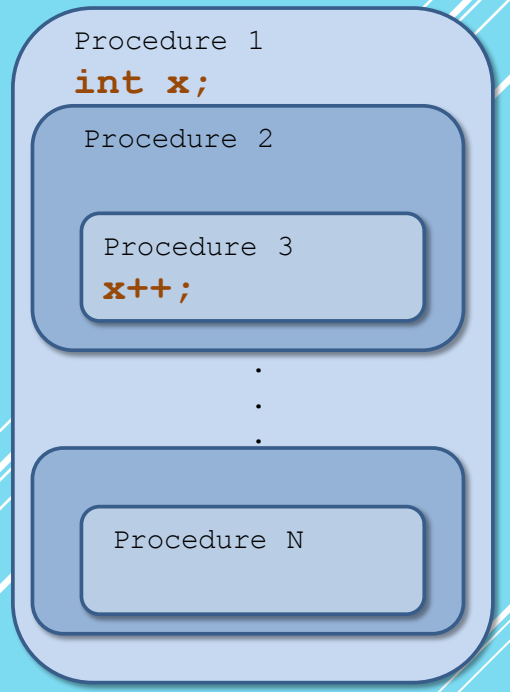
В многопоточном программировании область видимости получает дополнительное измерение – видимость между потоками. Исполнение параллельной программы перестаёт быть локальным!

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

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.

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.

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

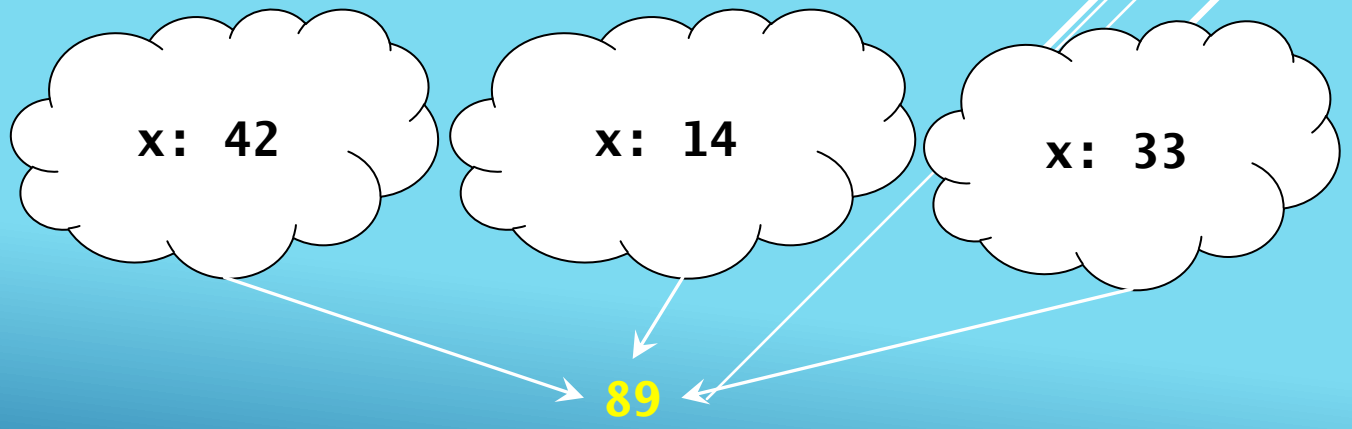
Изображение переменной – это её экземпляр. Потоки могут работать с переменной как с обычной нелокальной переменной.

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

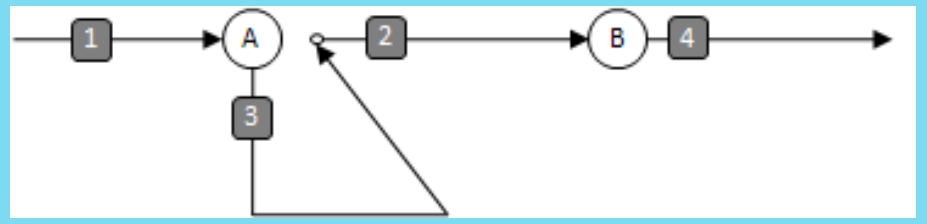
Система времени исполнения **Cilk Plus** координирует работу с изображениями переменной и объединяет их в точке объединения потоков (отсюда название - ***редукторы***).

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

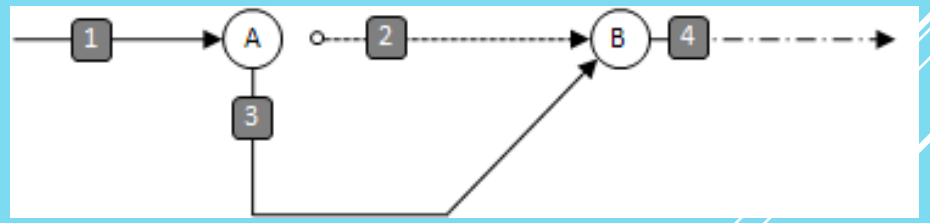
Редуктор суммирования:



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».

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 <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();
    }
};
```

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
```

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.

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;
```

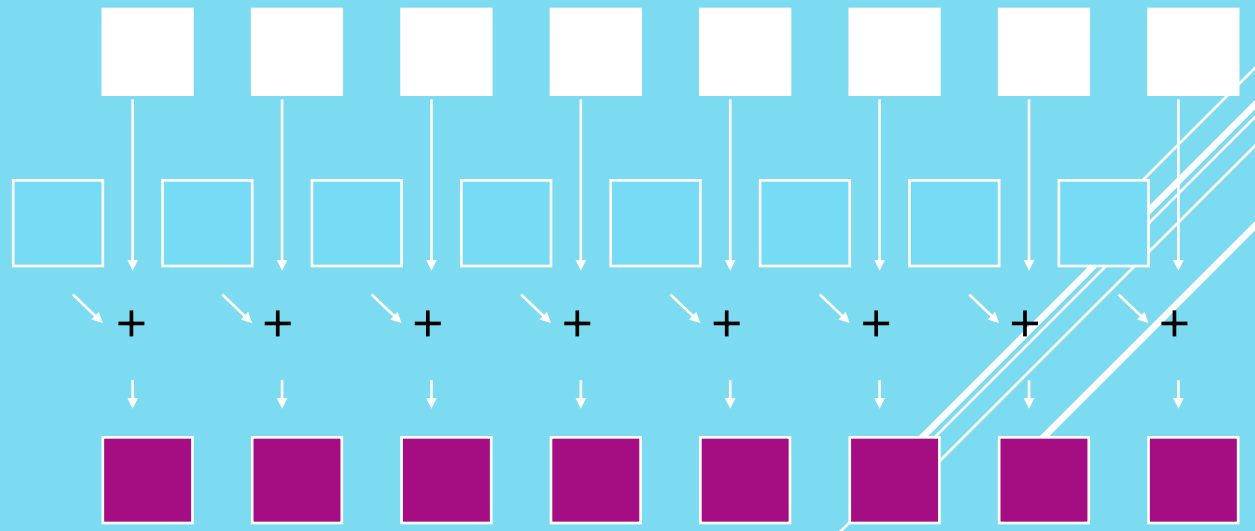
Ранги правой и левой частей должны совпадать. Допустимо использование скалярных величин:

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

Поэлементные векторные операции

Пример:

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



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`.

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]);
```

Ещё 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;
```

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.

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.

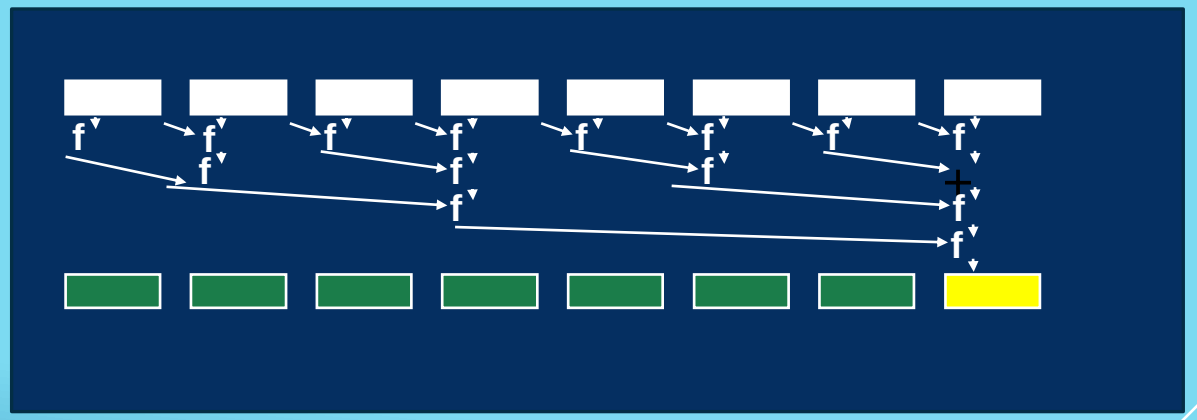
и другие.

Операции приведения (редукции)

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

Примеры:

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



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);
```

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

Используется для управления некоторыми параметрами системы исполнения Cilk.

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

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

Возвращает целое значение, показывающее исполнителя, который выполняет функцию.

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

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