



# ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В СТАНДАРТЕ OpenMP

*Общая сумма разума на планете есть  
величина постоянная, несмотря на  
постоянный прирост населения.*

*А. Блох*

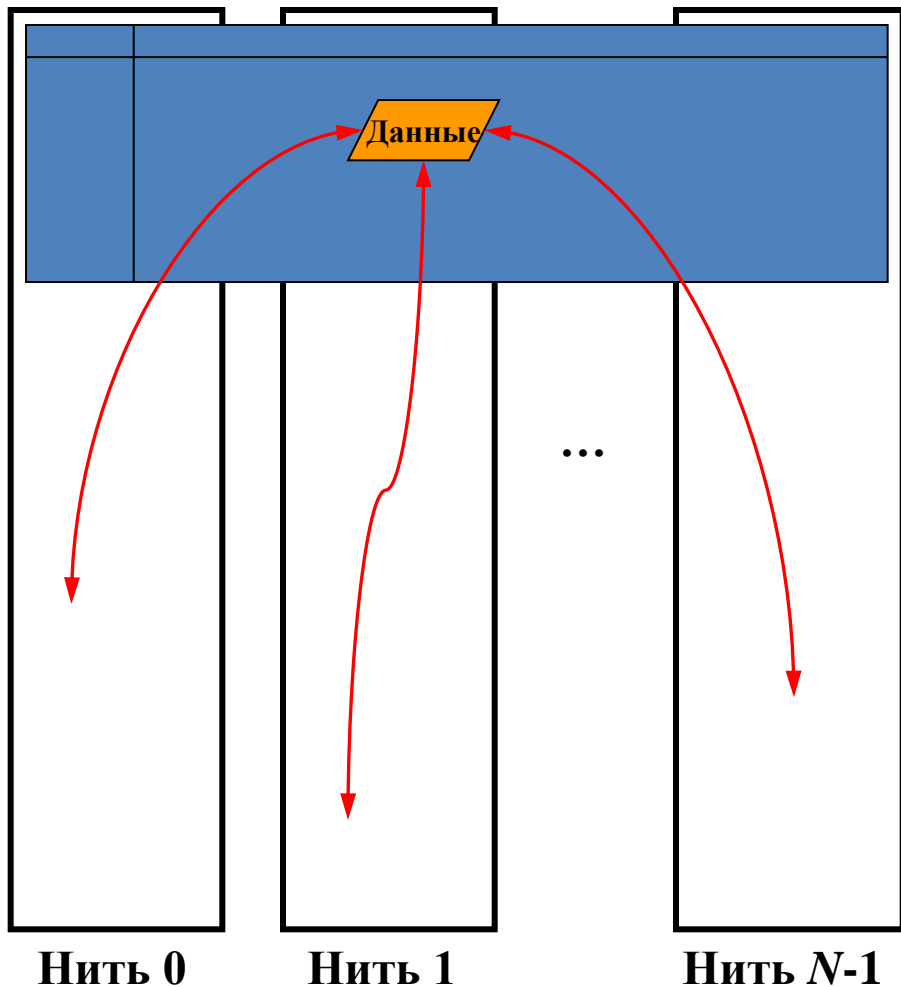
# Содержание

2

- ❑ Модель программирования в общей памяти
- ❑ Модель "пульсирующего" параллелизма FORK-JOIN
- ❑ Стандарт OpenMP
- ❑ Основные понятия и функции OpenMP

# Программирование в общей памяти

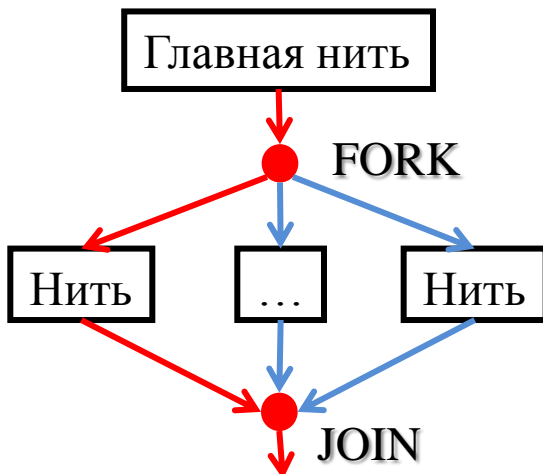
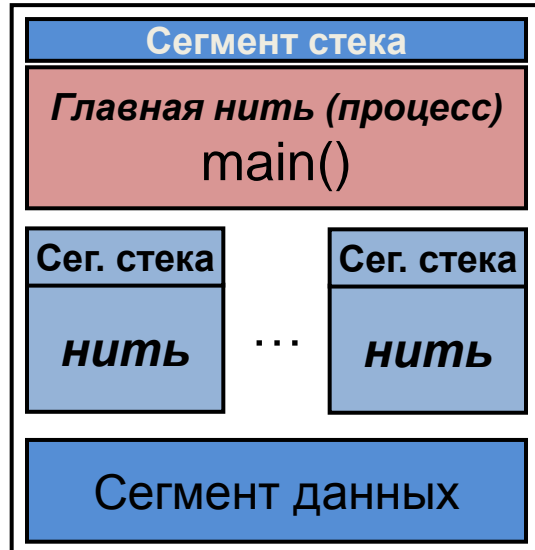
3



- *Параллельное приложение* состоит из нескольких *нитей*, выполняющихся одновременно.
- Нити разделяют общую память.
- Обмены между нитями осуществляются посредством чтения/записи данных в общей памяти.
- Нити выполняются на различных ядрах одного процессора.

# Модель FORK-JOIN

4



- ❑ Программа – полновесный *процесс*.
- ❑ Процесс может запускать *легковесные процессы (нити)*, выполняющиеся в фоновом режиме.
  - ▣ Процесс приложения – *главная нить*.
  - ▣ Нить может запускать другие нити в рамках процесса. Каждая нить имеет собственный сегмент стека.
  - ▣ Все нити процесса разделяют сегмент данных процесса.

# Стандарт OpenMP

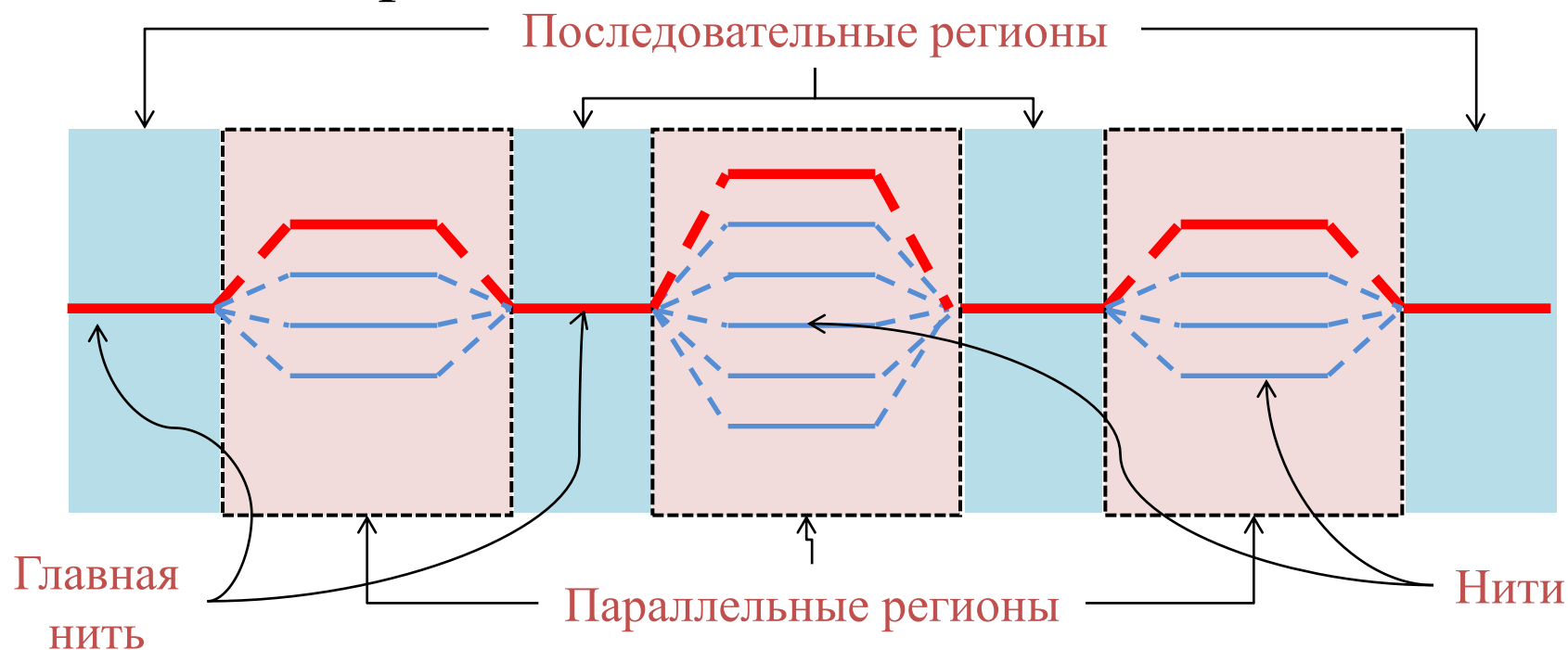
5

- *OpenMP (Open Multi-Processing)* – стандарт, реализующий модели программирования в общей памяти и Fork-Join.
- Стандарт содержит набор директив компилятора и спецификаций подпрограмм на языках C, C++ и FORTRAN.
- Стандарт реализуется разработчиками компиляторов для различных аппаратно-программных платформ (кластеры, персональные компьютеры, ..., Windows, Unix/Linux, ...).
- Разработчик стандарта – OpenMP Architecture Review Board ([www.openmp.org](http://www.openmp.org)).

# OpenMP-программа

6

- Главная нить (программа) порождает семейство дочерних нитей (сколько необходимо). Порождение и завершение осуществляется с помощью *директив компилятора*.



# Простая OpenMP-программа

7

Последовательный код

```
void main()  
{  
  
printf("Hello!\n");  
}
```

Результат

Hello!



Параллельный код

```
void main()  
{  
#pragma omp parallel  
{  
  
    printf("Hello!\n");  
  
}  
}
```

Результат

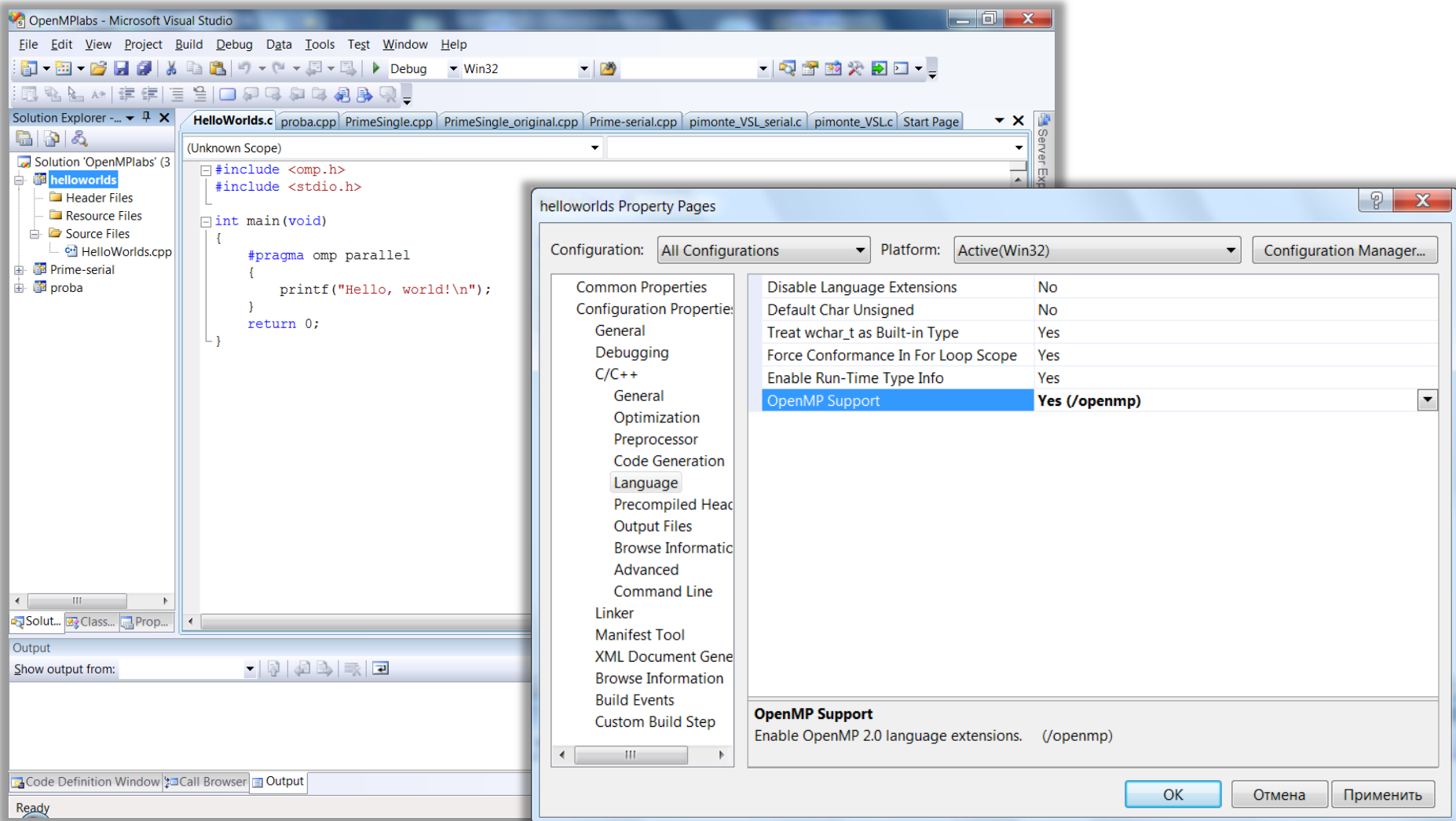
Hello!  
Hello!



(для 2-х нитей)

# Простая OpenMP-программа

8





# Преимущества OpenMP

9

- ❑ Поэтапное (инкрементное) распараллеливание
  - ▣ Можно распараллеливать последовательные программы поэтапно, не меняя их структуру.
- ❑ Единственность кода
  - ▣ Нет необходимости поддерживать последовательный и параллельный вариант программы, поскольку директивы игнорируются обычными компиляторами.
- ❑ Эффективность кода
  - ▣ Учет и использование возможностей систем с общей памятью.
- ❑ Мобильность кода
  - ▣ Поддержка языков C/C++, Fortran и ОС Windows, Unix/Linux.

# Директивы OpenMP

10

- Директивы OpenMP – директивы C/C++ компилятора `#pragma`.
  - ▣ параметр компиляции `/openmp`.
- Синтаксис директив OpenMP
  - ▣ `#pragma omp имя_директивы [параметры]`
- Примеры:
  - ▣ `#pragma omp parallel`
  - ▣ `#pragma omp for private(i, j) reduction(+: sum)`

# Функции библиотеки OpenMP

11

- Назначение функций библиотеки:
  - ▣ контроль и просмотр параметров OpenMP-программы
    - `omp_get_thread_num()` возвращает номер текущей нити
  - ▣ явная синхронизация нитей на базе "замков"
    - `omp_set_lock()` устанавливает "замок"
- Подключение библиотеки
  - ▣ `#include "omp.h"`

# Переменные окружения OpenMP

12

- Переменные окружения контролируют поведение приложения.
  - ▣ `OMP_NUM_THREADS` – количество нитей в параллельном регионе
  - ▣ `OMP_DYNAMIC` – разрешение или запрет динамического изменения количества нитей.
  - ▣ `OMP_NESTED` – разрешение или запрет вложенных параллельных регионов.
  - ▣ `OMP_SCHEDULE` – способ распределения итераций в цикле.
- Функции назначения параметров изменяют значения соответствующих переменных окружения.
- Макрос `_OPENMP` для условной компиляции отдельных участков исходного кода, характерных для параллельной версии программы.

# Область видимости переменных

13

- *Общая переменная (shared)* — глобальная по отношению к нити переменная; доступна для модификации всем нитям.
- *Частная переменная (private)* — локальная переменная нити; доступна для модификации только одной (создавшей ее) нити только на время выполнения этой нити.
- Видимость переменных по умолчанию:
  - переменные, определенные **вне** параллельной области — **общие**;
  - переменные, определенные **внутри** параллельной области — **частные**.
- Явное указание области видимости — параметры директив:
  - `#pragma omp parallel shared(buf)`
  - `#pragma omp for private(i, j)`

# Частные и общие переменные

14

```
void main()
{
    int a, b, c;
    ...
    #pragma omp parallel
    {
        int d, e;
        ...
    }
}
```

```
void main()
{
    int a, b, c;
    ...
    #pragma omp parallel shared(c) private(a)
    {
        int d, e;
        ...
    }
}
```

Частные переменные

Общие переменные

# Частные и общие переменные

15

```
void main()
{
    int rank;
    #pragma omp parallel
    {
        rank = omp_get_thread_num();
    }
    printf("%d\n", rank);
}
```

**Одно случайное число  
из диапазона  
0..OMP\_NUM\_THREADS-1**

```
void main()
{
    int rank;
    #pragma omp parallel
    {
        rank = omp_get_thread_num();
        printf("%d\n", rank);
    }
}
```

**OMP\_NUM\_THREADS  
случайных чисел  
(возможно, повторяющихся)  
из диапазона  
0..OMP\_NUM\_THREADS-1**

# Частные и общие переменные

16

```
void main()
{
    int rank;
#pragma omp parallel shared (rank)
    {
        rank = omp_get_thread_num();
        printf("%d\n", rank);
    }
}
```

```
void main()
{
    int rank;
#pragma omp parallel private (rank)
    {
        rank = omp_get_thread_num();
        printf("%d\n", rank);
    }
}
```

**OMP\_NUM\_THREADS**  
случайных чисел  
(возможно, повторяющихся)  
из диапазона  
 $0 \dots \text{OMP\_NUM\_THREADS}-1$

**OMP\_NUM\_THREADS**  
чисел из диапазона  
 $0 \dots \text{OMP\_NUM\_THREADS}-1$   
(без повторений,  
в случайном порядке)



# Распределение вычислений

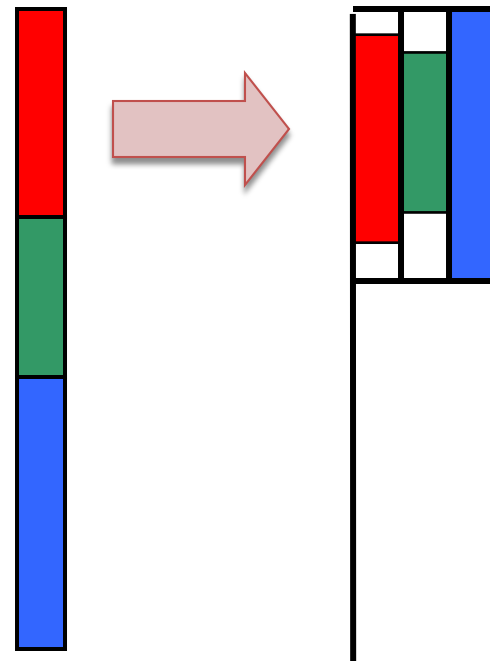
17

- Директивы распределения вычислений между нитями в параллельной области
  - ▣ `sections` – функциональное распараллеливание отдельных фрагментов кода
  - ▣ `single` и `master` – директивы для указания выполнения кода только одной нитью
  - ▣ `for` – распараллеливание циклов
- Начало выполнения директив по умолчанию не синхронизируется.
- Завершение директив по умолчанию является синхронным.

# Директива sections

18

```
#pragma omp parallel sections
{
    #pragma omp section
    Job1 ();
    #pragma omp section
    Job2 ();
    #pragma omp section
    Job3 ();
}
```



- Явное определение блоков кода, которые могут исполняться параллельно.
  - ▣ каждый фрагмент выполняется однократно
  - ▣ разные фрагменты выполняются разными нитями
  - ▣ завершение директивы синхронизируется.

# Директива `single`

19

- ❑ Определяет код, который выполняется только одной (первой пришедшей в данную точку) нитью.
  - ❑ Остальные нити пропускают соответствующий код и ожидают окончания его выполнения.
  - ❑ Если ожидание других нитей необязательно, может быть добавлен параметр `nowait`.

```
#pragma omp parallel
{
    #pragma omp single
        printf("Start Work #1.\n");
        Work1();
    #pragma omp single
        printf("Stop Work #1.\n");
    #pragma omp single nowait
        printf("Stop Work #1 and start Work #2.\n");
        Work2();
}
```

# Директива `master`

20

- ❑ Определяет код, который выполняется только одной главной нитью.
- ❑ Остальные нити пропускают соответствующий код, не ожидая окончания его выполнения.

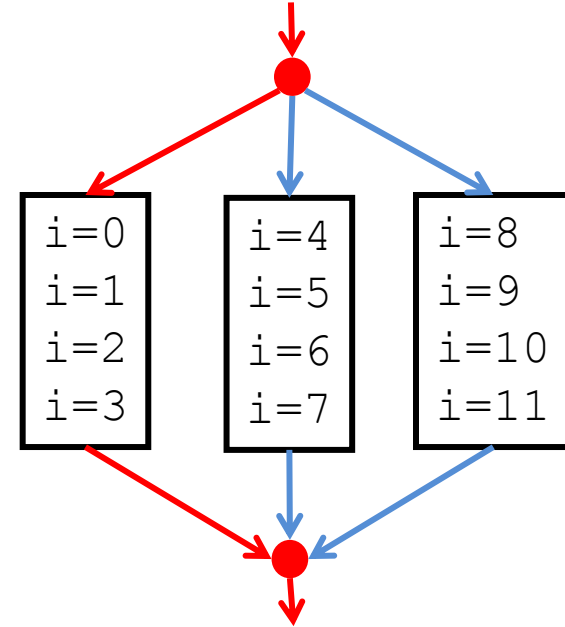
```
#pragma omp parallel
{
    #pragma omp master
        printf("Beginning Work1.\n");
        Work1();
    #pragma omp master
        printf("Finishing Work1.\n");
    #pragma omp master
        printf("Finished Work1 and beginning Work2.\n");
        Work2();
}
```

# Распараллеливание циклов

21

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<N; i++) {
        res[i] = big_calc();
    }
}
```

```
#pragma omp parallel for
for (i=0; i<N; i++) {
    res[i] = big_calc();
}
```



- ❑ Счетчик цикла по умолчанию является частной переменной.
- ❑ По умолчанию вычисления распределяются равномерно между нитями.

# Распараллеливание циклов

23

```
float scalar_product(float a[], float b[], int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(i=0; i<N; i++) {
        sum = sum + a[i] * b[i];
    }
    return sum;
}
```

- ❑ Бесконтрольное изменение нитями общих данных приводит к логическим ошибкам.

# Критическая секция в циклах

24

```
float scalar_product(float a[], float b[], int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(i=0; i<N; i++) {
        #pragma omp critical
        sum = sum + a[i] * b[i];
    }
    return sum;
}
```



- В любой момент времени код критической секции может быть выполнен только одной нитью.

# Редукция операций в циклах

25

- ❑ *Редукция* подразумевает определение для каждой нити частной переменной для вычисления "частичного" результата и автоматическое выполнение операции "слияния" частичных результатов.

```
float scalar_product(float a[], float b[], int N)
{
    float sum = 0.0;

    #pragma omp parallel for reduction(+:sum)
    for(i=0; i<N; i++) {
        sum = sum + a[i] * b[i];
    }
    return sum;
}
```

Опера ция	Нач. знач-е
+	0
*	1
-	0
^	0
&	~0
	0
&&	1
	0



# Редукция операций в циклах

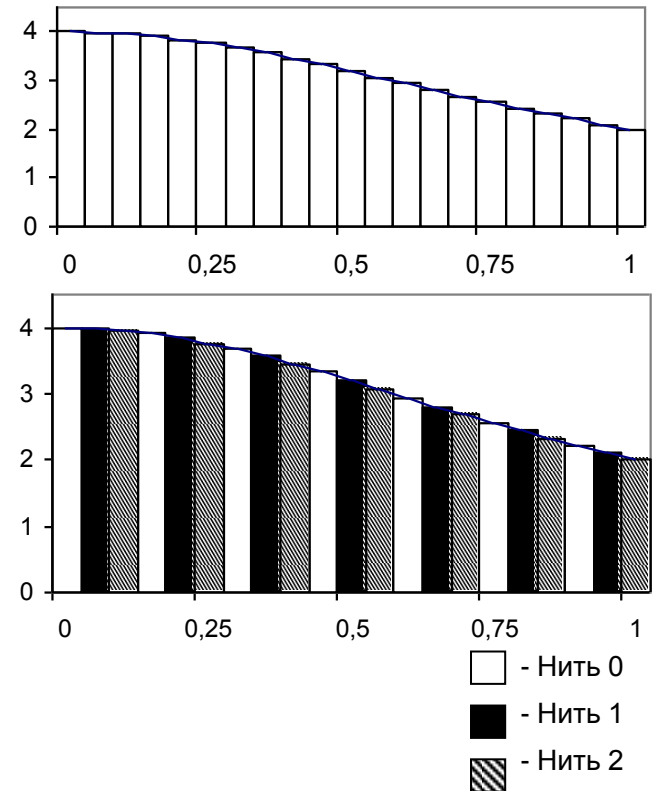
26

```
#include <stdio.h>
#include <time.h>
long long num_steps = 1000000000;
double step;
int main(int argc, char* argv[])
{
    clock_t start, stop;
    double x, pi, sum=0.0;
    int i;

    step = 1./(double)num_steps;
    start = clock();

    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i<num_steps; i++) {
        x = (i + 0.5)*step;
        sum = sum + 4.0/(1.+ x*x);
    }
    pi = sum*step;
    stop = clock();
    printf("PI=%15.12f\n", pi);
    printf("Time=%f sec.\n", ((double)(stop - start)/1000.0));
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



# Директива `for`

27

## ❑ Формат директивы

- ▣ `#pragma omp parallel for [clause ...]  
for (...)`

## ❑ Виды параметра `clause`

- ▣ `private(список_переменных)`
- ▣ `firstprivate(список_переменных)`
- ▣ `lastprivate(список_переменных)`
- ▣ `reduction(оператор: переменная)`
- ▣ `ordered`
- ▣ `nowait`
- ▣ `schedule(вид_распределения[, размер])`

# Параметр `firstprivate`

28

- ❑ Определяет частные переменные цикла `for`, которые в начале цикла принимают значения последовательной части программы.

```
myrank = omp_get_thread_num();  
#pragma omp parallel for firstprivate(myrank)  
for (i=0; i<N-1; i++) {  
    a[i] = b[i] + b[i+1] + myrank;  
    myrank = myrank + N % (i+1);  
}
```

# Параметр `lastprivate`

29

- Определяет частные переменные, которые по окончании цикла `for` принимают такие значения, как если бы цикл выполнялся последовательно.

```
#pragma omp parallel for lastprivate(i)
for (i=0; i<N-1; i++) {
    a[i] = b[i] + b[i+1];
}
// здесь i=N
```

# Параметр `ordered`

30

- Определяет код в теле цикла `for`, выполняемый в точности в том порядке, в каком он выполнялся бы при последовательном исполнении цикла.

```
#pragma omp for ordered schedule(dynamic)
for (i=start; i<stop; i+=step)
    Process(i);

void Process(int k)
{
    #pragma omp ordered
    printf(" %d", k);
}
```

# Параметр `nowait`

31

- Позволяет избежать неявного барьера при завершении директивы `for`.

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for nowait
    for (i=0; i<m; i++)
        y[i] = sqrt(z[i]);
}
```

# Распределение итераций цикла

32

- Распределение итераций в директиве `for` регулируется параметром `schedule` (вид\_распределения [, размер])
  - `static` – итерации делятся на блоки по размер итераций и статически разделяются между потоками; если параметр `размер` не определен, итерации делятся между потоками равномерно и непрерывно
  - `dynamic` – распределение итерационных блоков осуществляется динамически (по умолчанию `размер=1`)
  - `guided` – размер итерационного блока уменьшается экспоненциально при каждом распределении; `размер` определяет минимальный размер блока (по умолчанию `размер=1`)
  - `runtime` – правило распределения определяется переменной `OMP_SCHEDULE` (при использовании `runtime` параметр `размер` задаваться не должен)

# Пример

33

```
// Объем работы в итерациях предсказуем и примерно одинаков  
#pragma omp parallel for schedule(static)  
for(i=0; i<n; i++) {  
    invariant_amount_of_work(i);  
}
```

```
// Объем работы в итерациях может существенно различаться  
// или непредсказуем  
#pragma omp parallel for schedule(dynamic)  
for(i=0; i<n; i++) {  
    unpredictable_amount_of_work(i);  
}
```



# Пример

34

```
// Нити подходят к точке распределения итераций
// в разное время, объем работы в итерациях
// предсказуем и примерно одинаков
#pragma omp parallel
{
    #pragma omp sections nowait
    {
        ...
    }
    #pragma omp for schedule(guided)
    for(i=0; i<n; i++) {
        invariant_amount_of_work(i);
    }
}
```

# Синхронизация вычислений

35

- Директивы явной синхронизации
  - ▣ `critical`
  - ▣ `barrier`
  - ▣ `atomic`
- Директива неявной синхронизации
  - ▣ `#pragma omp parallel`

# Директива `critical`

36

- ❑ Определяет *критическую секцию* – участок кода, выполняемый одновременно не более чем одной НИТЬЮ.

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
#pragma omp critical (Xaxis_critical_section)
    x_next = Queue_Remove(x);
    Process(x_next);
#pragma omp critical (Yaxis_critical_section)
    y_next = Queue_Remove(y);
    Process(y_next);
}
```

# Директива `atomic`

37

- ❑ Определяет *критическую секцию* для одного оператора вида
  - ❑ `x++` и `++x`
  - ❑ `x--` и `--x`
  - ❑ `x+=выражение`, `x-=выражение` и др.

```
extern float a[], *p = a, b;
```

```
// Предохранение от гонок данных  
// при обновлении несколькими нитями  
#pragma omp atomic  
a[index[i]] += b;
```

```
// Предохранение от гонок данных  
// при обновлении несколькими нитями  
#pragma omp atomic  
p[i] -= 1.0f;
```

# Директива `barrier`

38

- ❑ Определяет *барьер* – точку в программе, которую должна достигнуть каждая нить, чтобы все нити продолжили вычисления.

```
#pragma omp parallel shared (A, TmpRes, FinalRes)
{
    DoSomeWork(A, TmpRes);
    printf("Processed A into TmpRes\n");
#pragma omp barrier
    DoSomeWork(TmpRes, FinalRes);
    printf("Processed B into C\n");
}
```

```
// Директива должна быть частью структурного блока
if (x!=0) {
    #pragma omp barrier
    ...
}
```

# Директива `barrier`

39

```
int main()
{
sub1(2);
sub2(2);
sub3(2);
}
void sub1(int n)
{
int i;
#pragma omp parallel private(i) shared(n)
{
#pragma omp for
for (i=0; i<n; i++)
sub2(i);
}
}
void sub2(int k)
{
#pragma omp parallel shared(k)
sub3(k);
}
void sub3(int n)
{
work(n);
#pragma omp barrier
work(n);
}
```

# Директивы и параметры

40

Параметр	Директива					
	parallel	for	sections	single	parallel for	parallel sections
if	✓				✓	✓
private	✓	✓	✓	✓	✓	✓
shared	✓	✓			✓	✓
default	✓				✓	✓
firstprivate	✓	✓	✓	✓	✓	✓
lastprivate		✓	✓		✓	✓
reduction	✓	✓	✓		✓	✓
copyin	✓				✓	✓
schedule		✓			✓	
ordered		✓			✓	
nowait		✓	✓	✓		

# Время работы

41

```
double start;  
double end;  
  
start = omp_get_wtime();  
// Работа  
end = omp_get_wtime();  
printf("Work took %f sec. time.\n", end-start);
```



# Количество нитей

42

**// Неверно**

```
np = omp_get_num_threads(); // Здесь еще не выполнен FORK
#pragma omp parallel for schedule(static)
for (i=0; i<np; i++)
    work(i);
```

**// Верно**

```
#pragma omp parallel private(i)
{
    i = omp_get_thread_num();
    work(i);
}
```

# Заключение

46

- ❑ Модель программирования в общей памяти
- ❑ Модель "пульсирующего" параллелизма FORK-JOIN
- ❑ Стандарт OpenMP
- ❑ Основные понятия и функции OpenMP

# Информационные ресурсы

47

- Что такое OpenMP  
[http://parallel.ru/tech/tech\\_dev/openmp.html](http://parallel.ru/tech/tech_dev/openmp.html)
- Introduction to OpenMP  
[www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html](http://www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html)
- Chandra R., Menon R., et al. Parallel Programming in OpenMP. Morgan Kaufmann, 2000.
- Quinn M.J. Parallel Programming in C with MPI and OpenMP. McGraw-Hill, 2004.
- [www.openmp.org](http://www.openmp.org)