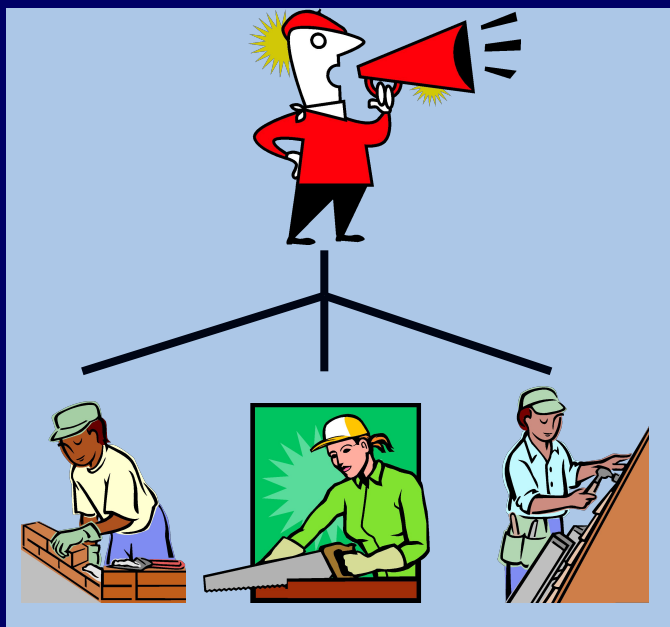


# Параллельное программирование в стандарте OpenMP

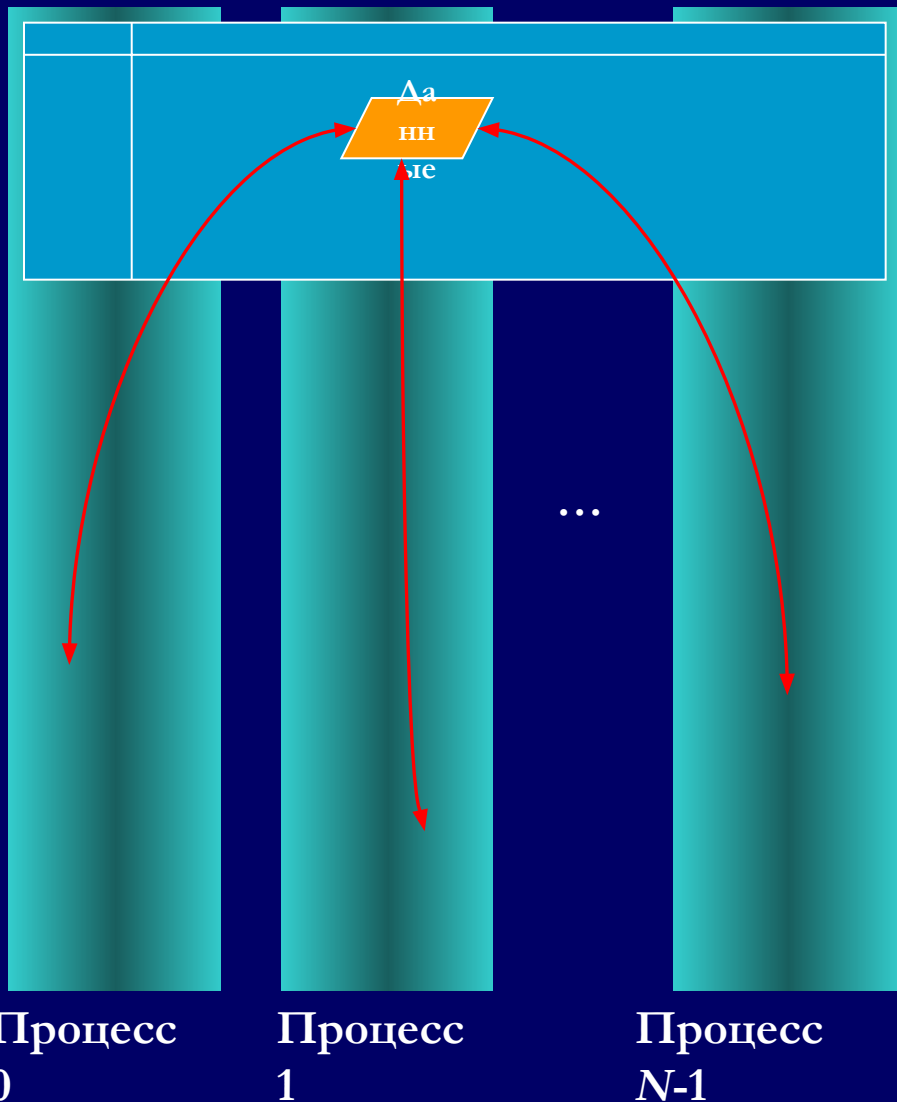


# Содержание

---

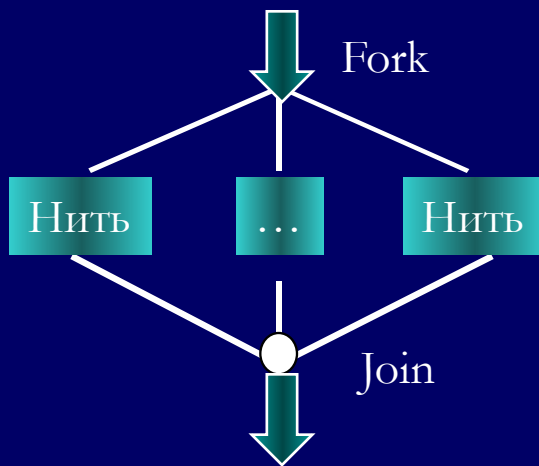
- Модель программирования в общей памяти
- Модель FORK-JOIN
- Стандарт OpenMP
- Основные понятия и функции OpenMP

# Программирование в общей памяти



- *Параллельное приложение* состоит из нескольких *процессов*, выполняющихся одновременно.
- Процессы разделяют общую память.
- Обмены между процессами осуществляются посредством чтения/записи данных в общей памяти.
- Процессы могут выполняться как на одном и том же, так и на разных процессорах.

# Модель FORK-JOIN



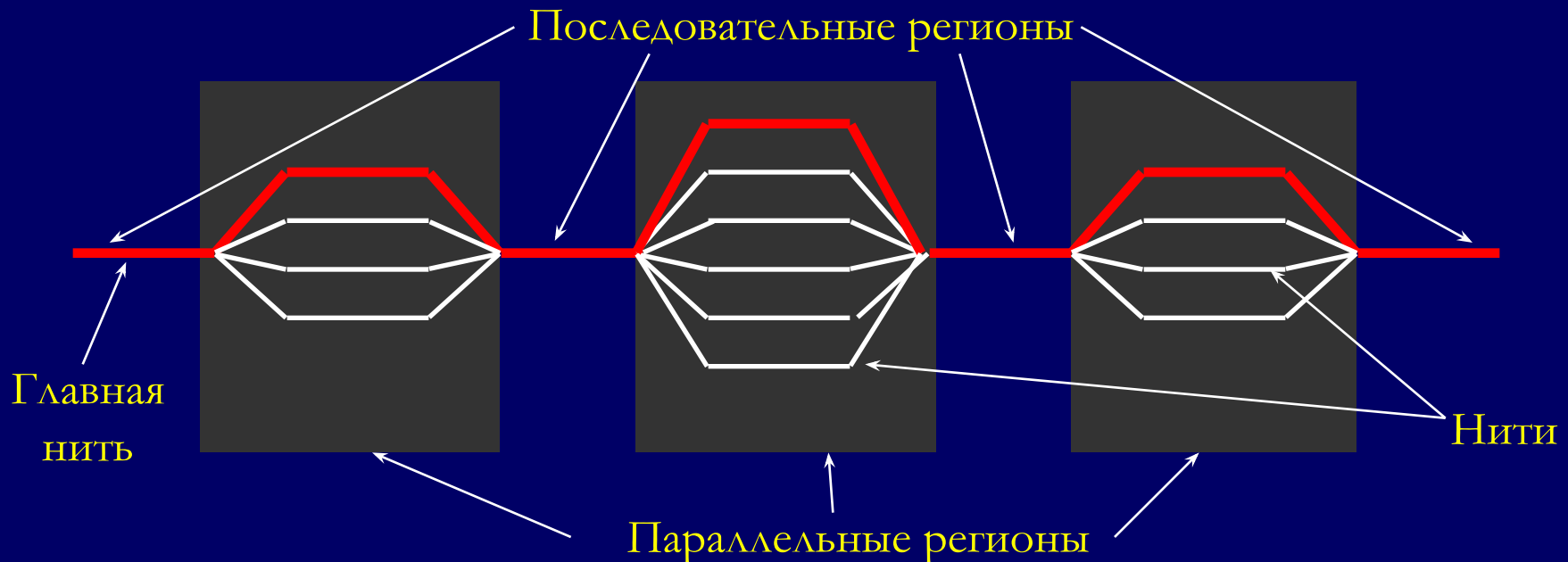
- Современные операционные системы поддерживают *полновесные процессы* (программы) и *легковесные процессы (нити)*.
  - Процесс – *главная нить*.
  - Нить может запускать другие нити в рамках процесса. Каждая нить имеет собственный сегмент стека.
  - Все нити процесса разделяют сегмент данных процесса.

# Стандарт OpenMP

- *OpenMP* – стандарт, реализующий модели программирования в общей памяти и Fork-Join.
- Стандарт представляет собой набор директив компилятора и спецификаций подпрограмм для на языках C, C++ и FORTRAN.
- Стандарт реализуется разработчиками компиляторов для различных аппаратно-программных платформ (кластеры, персональные компьютеры, ..., Windows, Unix/Linux, ...).

# Структура OpenMP-программы

- Главная нить (программа) порождает семейство дочерних нитей (сколько необходимо). Порождение и завершение осуществляется с помощью директив компилятора.
- Преобразование последовательной программы в параллельную может происходить "инкрементно".



# Директивы OpenMP

- Директивы OpenMP – директивы C/C++ компилятора `#pragma`.
  - Для использования директив необходимо установить соответствующие параметры компилятора (обычно `-openmp`).
- Синтаксис директив OpenMP:  
`#pragma omp имя_директивы [параметры]`
- Примеры:  
`#pragma omp parallel`  
`#pragma omp for private(i, j) reduction(+: sum)`

# Функции библиотеки OpenMP

- 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"
- Для использования функций необходимо подключить библиотеку
  - `#include "omp.h"`



# Переменные окружения OpenMP

- 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.

# Директивы определения параллельных фрагментов

- `#pragma omp parallel`  
Определяет блок кода, который будет выполнен всеми созданными на входе в этот блок нитями.
- `#pragma omp for`  
Определяет цикл, итерации которого должны выполняться одновременно несколькими нитями.
- `#pragma omp section`  
Определяет множество блоков кода, каждый из которых будет выполняться только одной из созданных на входе в этот блок нитью.
- `#pragma omp master`  
Определяет блок кода, который будет выполнен только главной нитью.

# Простая программа на OpenMP

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)

# Директива `parallel for`

- OpenMP поддерживает *редукцию* вычислительных операций, выполняемых в циклах. Редукция подразумевает определение для каждой нити частной переменной для вычисления "частичного" результата и автоматическое выполнение операции "слияния" частичных результатов.

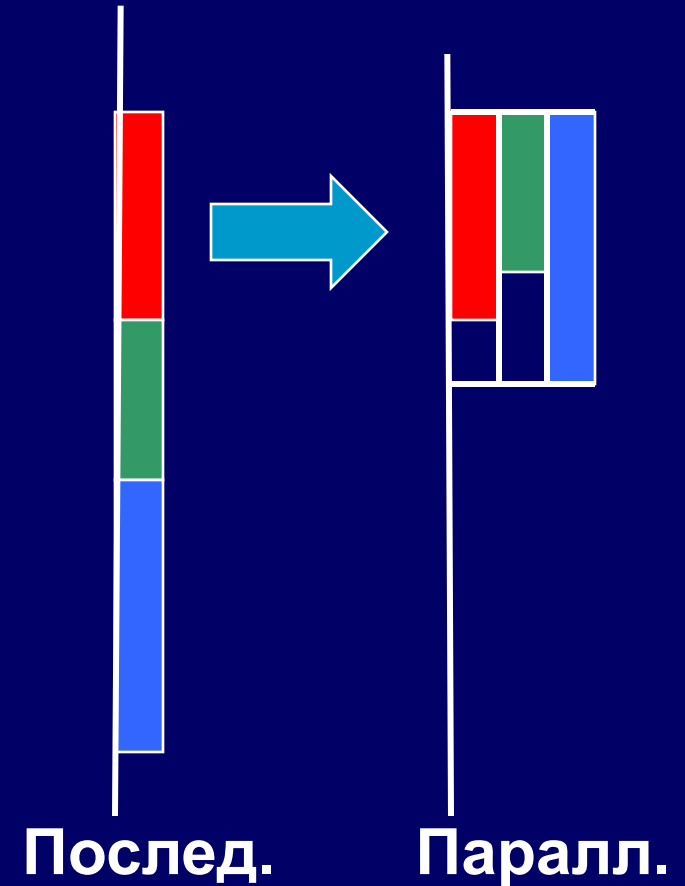
```
#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++) {
    sum += a[i] * b[i];
}
```

Операция	Нач. значение
+	0
*	1
-	0
^	0
&	~0
	0
&&	1
	0

# Параллельные секции

- Явное определение блоков кода, которые могут исполняться параллельно.

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



# Область видимости переменных

- *Общая переменная (shared)* – доступна для модификации всем нитям.
- *Частная переменная (private)* – доступна для модификации только одной (создавшей ее) нити только на время выполнения этой нити.
- Правила видимости переменных:
  - все переменные, определенные вне параллельной области – **общие**;
  - все переменные, определенные внутри параллельной области – **частные**.

# Общие и частные переменные

```
void main()
```

```
{
```

```
int a, b, c;
```

```
...
```

```
#pragma omp parallel
```

```
{
```

```
int d, e;
```

```
...
```

```
}
```

```
}
```

Общи

e

Частные

# Явное указание области видимости

- Для явного указания области видимости используются следующие параметры директив:
  - `shared()` – общие переменные
  - `private()` – частные переменные
- Примеры:
  - `#pragma omp parallel shared(buf)`
  - `#pragma omp for private(i, j)`



# Общие и частные переменные

```
void main()
```

```
{
```

```
int a, b, c;
```

```
...
```

```
#pragma omp parallel shared(a) private(b)
```

```
{
```

```
int d, e;
```

```
...
```

```
}
```

```
}
```

Общи

е

Частные

# Общие и частные переменные

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

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

# Общие и частные переменные

```
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);
    }
}
```

# Директивы синхронизации

- `#pragma omp master`  
Определяет блок кода, который будет выполнен только главной нитью.
- `#pragma omp critical`  
Определяет блок кода, который не должен выполняться одновременно двумя или более нитями.
- `#pragma omp barrier`  
Определяет точку барьерной синхронизации, в которой каждая нить дожидается всех остальных.
- `#pragma omp atomic`  
Определяет переменную в левой части оператора "атомарного" присваивания, которая должна корректно обновляться несколькими нитями.
- `#pragma omp flush`  
Явно определяет точку, в которой обеспечивается одинаковый вид памяти для всех нитей.

# #pragma omp master

- Определяет блок кода, который будет выполнен только главной нитью. Не подразумевает барьера для других нитей.

```
#pragma omp parallel
{
    DoSomeJob1(omp_get_thread_num());
#pragma master
{
    printf("Job #1 done\n");
}
    DoSomeJob2(omp_get_thread_num());
#pragma master
{
    printf("Job #2 done\n");
}
}
```

# #pragma omp critical

- Определяет блок кода, который не должен выполняться одновременно двумя или более нитями.

```
float dot_prod(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;
}
```

```
float dot_prod(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 += a[i] * b[i];
    }
    return sum;
}
```