

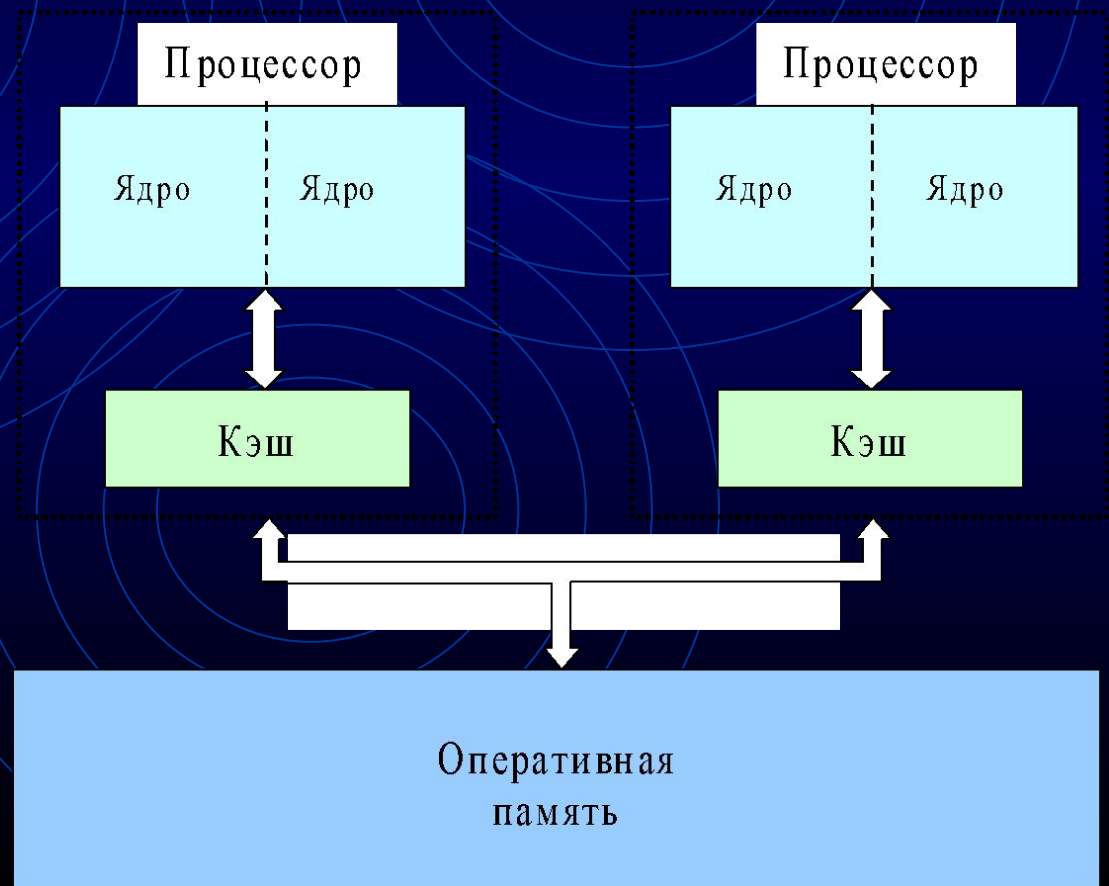
OpenMP

Параллельное
программирование для
многоядерных систем

<http://www.openmp.org>.

Концепция OpenMP

Интерфейс OpenMP задуман как стандарт параллельного программирования для многопроцессорных систем с общей памятью



Положительные качества OpenMP

Поэтапное распараллеливание

Можно распараллеливать последовательные программы поэтапно, не меняя их структуру

Единственность разрабатываемого кода

Нет необходимости поддерживать последовательный и параллельный вариант программы, поскольку директивы игнорируются обычными компиляторами (в общем случае)

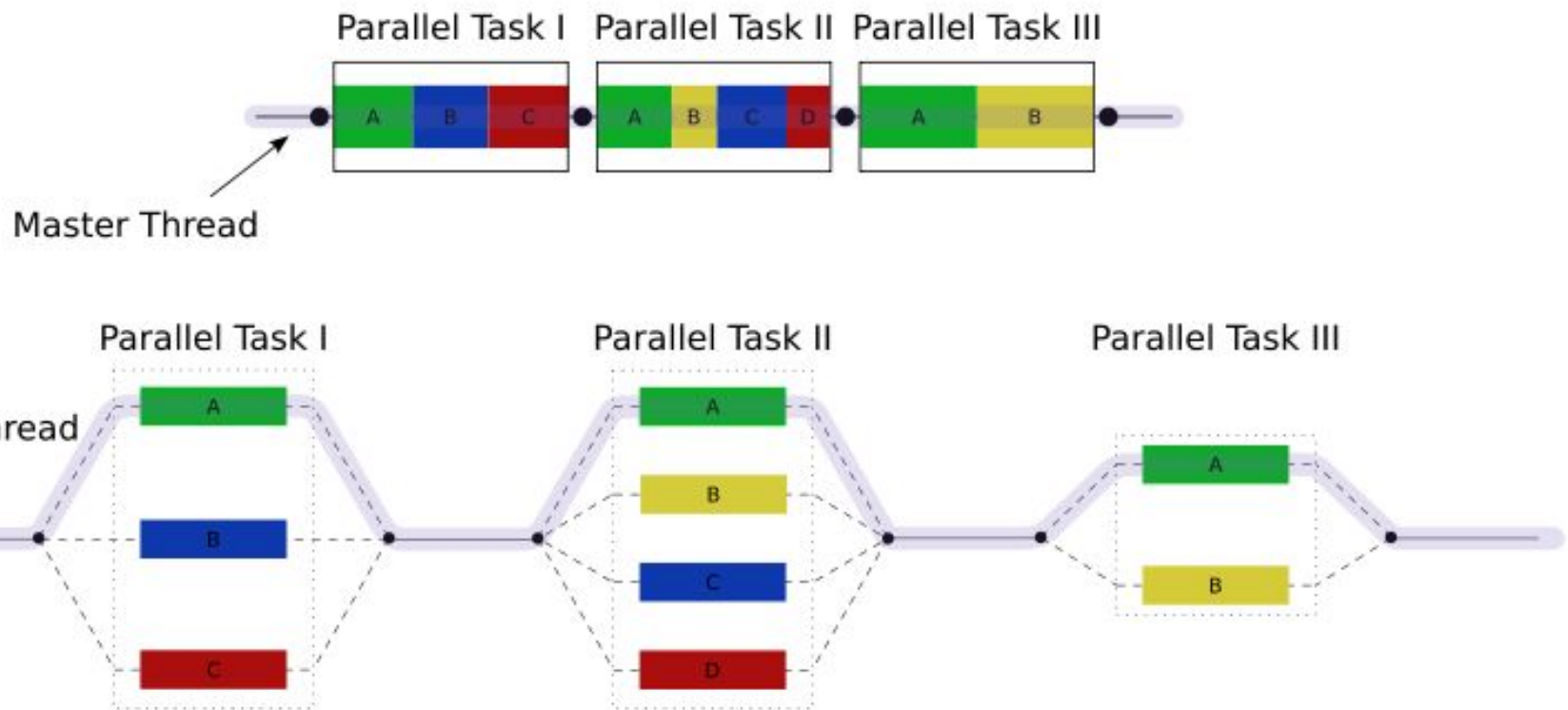
Переносимость

Поддержка большим числом компиляторов под разные платформы и ОС, стандарт для распространенных языков C/C++, Fortran

Принципы организации параллелизма

Использование потоков на общем адресном пространстве

Пульсирующий (fork-join) параллелизм



Принципы организации параллелизма

- При выполнении обычного кода (вне параллельных областей) программа исполняется одним потоком (master thread)
- При появлении директивы `#parallel` происходит создание “команды” потоков для параллельного выполнения вычислений
- После выхода из области действия директивы `#parallel` происходит синхронизация, все потоки, кроме master, уничтожаются
- Продолжается последовательное выполнение кода (до очередного появления директивы `#parallel`)

Когда следует использовать технология Open MP

- Целевая платформа является многопроцессорной или многоядерной
- Выполнение циклов нужно распараллелить
- Перед выпуском приложения нужно повысить его быстродействие
- Параллельное приложение должно быть кроссплатформенным

Когда эффективно использовать технологию Open MP

- Параллельная версия настигает по быстродействию последовательную или при **большом количестве итераций** или при **большом объеме вычислений** внутри параллельных фрагментов кода

Настройки компилятора Microsoft Visual Studio

Project

Property Pages

C/C++

Language

OpenMP Support

Yes

Средства openMP

Категории:

- Функции времени выполнения
- Функции инициализации/завершения
- Переменные среды окружения
- Параллельные регионы
- Распределение работ и диспетчеризация
- Синхронизация и блокировка

Структура программы

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

```
int main() {
```

```
    // по умолчанию кол-во потоков:
```

```
    int numTh = omp_get_num_threads();
```

```
    // сами ставим кол-во потоков
```

```
    omp_set_num_threads(4);
```

```
    #pragma omp parallel
```

```
    ...
```

```
    return 0;
```

```
}
```

Простейшая программа

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char * argv[]) {
    #pragma omp parallel
    {
        printf("Hello world\n");
    }
    printf("\nset num_threads=16\n");
    omp_set_num_threads(16);
    #pragma omp parallel
    {
        printf("Hello world\n");
    }
    return 0;
}
```


Простейшие директивы OpenMP

```
#pragma omp parallel
```

```
{
```

```
    <КОД>
```

```
}
```

```
// выполнится столько раз, сколько потоков
```

```
#pragma omp parallel for
```

```
for (int i=0; i<n; i++){
```

```
    <КОД>
```

```
}
```

```
// выполнится n раз с разделением нагрузки между потокам
```

Ограничения на оператор for OpenMP

1. Переменная цикла должна иметь тип `integer`.
2. Цикл должен являться базовым блоком и не может использовать `goto` и `break` (за исключением оператора `exit`, который завершает все приложение).
3. Инкрементная часть цикла `for` должна являться либо целочисленным сложением, либо целочисленным вычитанием.
4. Если используется операция сравнения `<` или `<=`, переменная цикла должна увеличиваться при каждой итерации, а при использовании операции `>` или `>=` переменная цикла должна уменьшаться.

Пример 1

```
#pragma omp parallel num_threads(2)
```

```
  for (int i = 0; i < 10; i++)
```

```
    myFunc();
```

```
// при запуске будет выполняться по одному разу в
```

```
// каждом потоке, и вместо ожидаемых 10 раз
```

```
// функция myFunc будет вызвана 20 раз.
```

```
#pragma omp parallel for num_threads(2)
```

```
  for (int i = 0; i < 10; i++)
```

```
    myFunc();
```

```
// цикл будет выполнен 10 раз, разделенный
```

```
// между двумя потоками
```

Пример 1-а (parallel без указания количества потоков)

```
#pragma omp parallel  
for (int i = 0; i < 10; i++)  
    myFunc();
```

// при запуске будет выполняться по одному разу в
// каждом потоке, и вместо ожидаемых 10 раз
// функция myFunc будет вызвана 20 раз при двух процессорах

```
#pragma omp parallel for  
for (int i = 0; i < 10; i++)  
    myFunc();
```

// цикл будет выполнен 10 раз, разделенный
// между двумя потоками на двухпроцессорном компьютере

Пример 2 (ошибка!)

```
#pragma omp parallel num_threads(2)
{
    ... // N строк кода
    #pragma omp parallel for
    for (int i = 0; i < 10; i++) {
        myFunc();
    }
}
// в двух потоках выполнятся два
// параллельных цикла
// функция myFunc будет вызвана 20 раз.
```

Пример 2 - правильно

```
#pragma omp parallel num_threads(2)
{
    ... // N строк кода
    #pragma omp for
    for (int i = 0; i < 10; i++) {
        myFunc();
    }
}
```

Ограничение на переопределение количества потоков

Количество потоков нельзя переопределять внутри параллельной секции. Это приводит к ошибкам во время выполнения программы и ее аварийному завершению.

```
#pragma omp parallel
{
  omp_set_num_threads(2); // ошибка !!!!
  #pragma omp for
  for (int i = 0; i < 10; i++)
    myFunc();
}
```

Планирование и разбиение циклов

Static scheduling – цикл делится на фрагменты
одинакового размера

Dynamic scheduling – фрагменты помещаются
в очередь, освободившийся
поток будет брать
следующую «порцию».

Guided scheduling == Dynamic scheduling
с изменяемым размером
фрагмента в процессе выполнения

Планирование и разбиение циклов (Static scheduling - пример работы)

```
int s;  
#pragma omp parallel for private (s)  
for (int i=0; i<10; i++){  
    s = omp_get_thread_num();  
    Sleep(1000*i);  
    printf("Hello world from thread = %d\n",s);  
}
```

Планирование и разбиение циклов (Static scheduling - пример работы)

c:\MY_PROGRAMMS\OpenMP_01\Debug\OpenMP_01.exe

```
Hello world from thread = 0  
Hello world from thread = 0  
Hello world from thread = 0  
Hello world from thread = 1  
Hello world from thread = 0  
Hello world from thread = 0  
Hello world from thread = 1  
Hello world from thread = 1  
Hello world from thread = 1  
Hello world from thread = 1
```

Step(1000*);

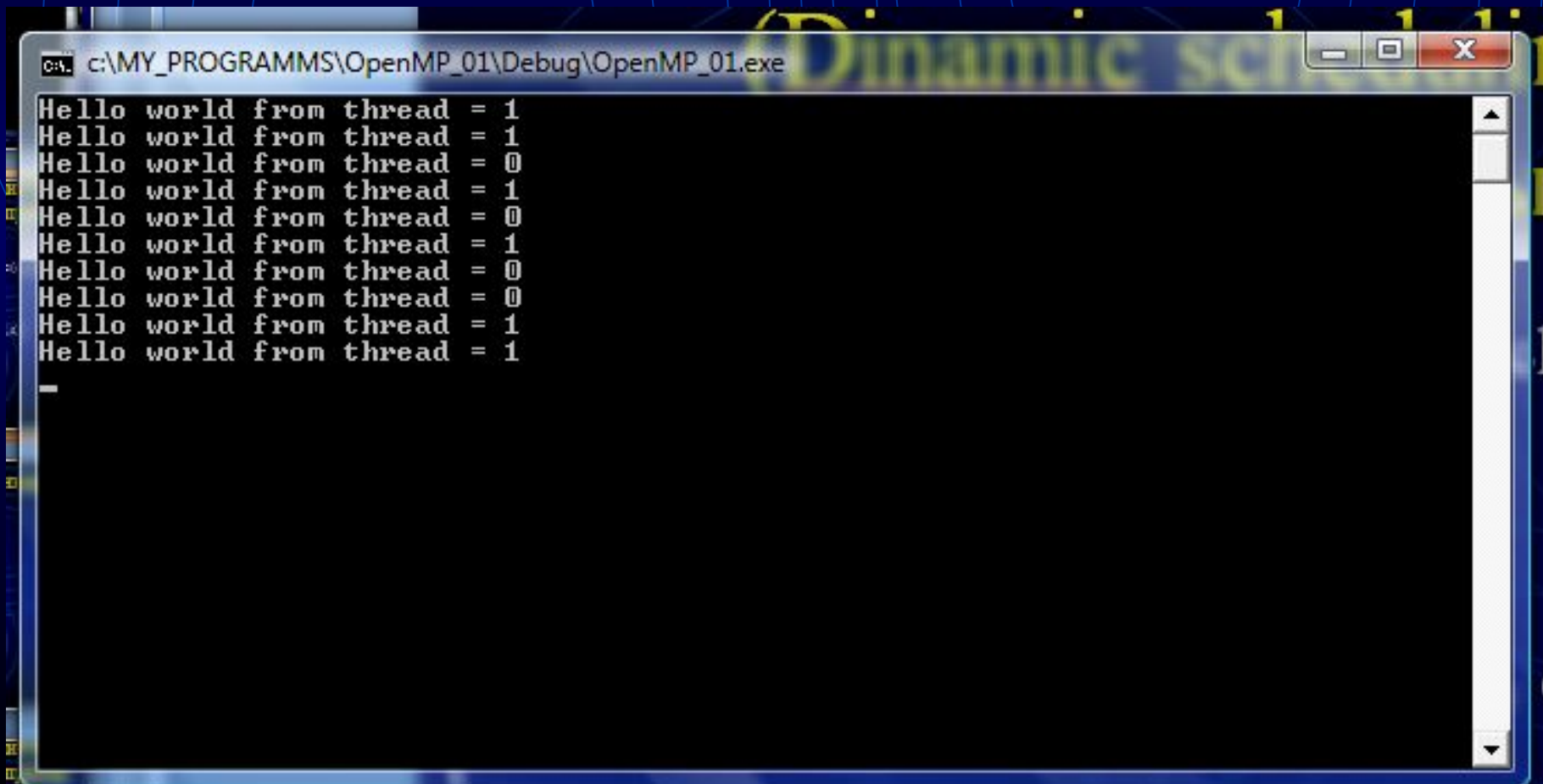
Планирование и разбиение циклов (Dynamic scheduling - пример работы)

```
int s;  
#pragma omp parallel for private (s)schedule(dynamic, 2)  
for (int i=0; i<10; i++){  
    s = omp_get_thread_num();  
    Sleep(1000*i);  
    printf("Hello world from thread = %d\n",s);  
}
```

Планирование и разбиение циклов

Dynamic scheduling - пример работы

`schedule(dynamic, 2)`



```
c:\MY_PROGRAMMS\OpenMP_01\Debug\OpenMP_01.exe
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 0
Hello world from thread = 1
Hello world from thread = 0
Hello world from thread = 1
Hello world from thread = 0
Hello world from thread = 0
Hello world from thread = 1
Hello world from thread = 1
```


Планирование и разбиение циклов

Dynamic scheduling - пример работы

`schedule(dynamic, 1)`

c:\MY_PROGRAMMS\OpenMP_01\Debug\OpenMP_01.exe

```
Hello world from thread = 1  
Hello world from thread = 0  
Hello world from thread = 1  
Hello world from thread = 0  
Hello world from thread = 1  
Hello world from thread = 0  
Hello world from thread = 1  
Hello world from thread = 0  
Hello world from thread = 1  
Hello world from thread = 0
```

Dinamic scheduling - пример работы для быстрых операций во втором потоке и медленных в первом

```
c:\MY_PROGRAMMS\OpenMP_01\Debug\OpenMP_01.exe
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 0
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 0
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 1
Hello world from thread = 0
Hello world from thread = 1
```

**Вывод данных в консоль в
параллельных потоках
- операция ОЧЕНЬ опасная!!!**

Операция вывода строки на экран не является атомарной. Следовательно, два потока будут выводить свои символы одновременно.

Возможны гонки данных

Вывод данных в консоль в параллельных потоках надо оградить блокировкой

```
#pragma omp parallel num_threads(2)
{
    #pragma omp critical
    {
        printf("Hello World \n");
    }
}
```

Гонки данных

Незащищенный доступ к общей памяти

```
double data;  
#pragma omp parallel for  
    for (int i = 0; i < 10; i++) {  
        data=array[i];  
        array[i] = func(data);  
        // гонки данных!!  
        // переменная data    одна для всех потоков  
    }
```

Гонки данных

Незащищенный доступ к общей памяти

ПРИЧИНА:

все глобальные переменные в OpenMP считаются `shared` по умолчанию/

РЕШЕНИЕ (первый вариант):

просто объявлять соответствующие переменные как локальные переменные в параллельных секциях.

Приватные переменные

- метод борьбы с гонками данных

```
double data;  
#pragma omp parallel for  
  for (int i = 0; i < 10; i++) {  
    data=array[i];            ЗАМЕНИМ КОД  
    array[i] = func(data);  
    // ГОНКИ ДАННЫХ!!  
  }
```

Локальные переменные становятся приватными – первый вариант решения

```
// double data; // удалили внешнее объявление!  
#pragma omp parallel for  
    for (int i = 0; i < 10; i++) {  
        double data=array[i];  
        array[i] = func(data);  
// переменная, объявленная внутри  
// параллельной конструкции – приватная  
    }
```


Явно указанные приватные переменные — решение, эквивалентное по результату

```
double data;  
#pragma omp parallel for private(data)  
    for (int i = 0; i < 10; i++) {  
        data=array[i];  
        // переменная объявлена приватной в потоке  
        // гонки данных отсутствуют!!  
        array[i] = func(data);  
    }
```

Приватные переменные — ВОЗМОЖНЫЕ ОШИБКИ

1. При входе в поток для переменных, являющиеся параметрами выражений `private`, создаются **локальные копии**. Эти копии являются **неинициализированными** по умолчанию.

Следовательно, любая попытка работы с ними без предварительной инициализации приведет к ошибке во время выполнения программы.

2. **Забытое выражение `private` и работа с переменной вне параллельной секции**. Значения таких переменных после соответствующей параллельной секции являются **непредсказуемыми**.

Приватные переменные – ограничения

Переменная в выражении `private` не должна иметь ссылочный тип.

Причина ограничения очевидна – если переменная будет указателем, то каждый поток получит по локальной копии этого указателя, и в результате все потоки будут работать через него с общей памятью.

Правила разделения переменных (1)

Неявное правило 1: Все переменные, определенные вне `omp parallel`, являются глобальными для всех потоков

Неявное правило 2: Все переменные, определенные внутри `omp parallel`, являются локальными для каждого потока

Неявное исключение: В прагме `omp for`, счетчик цикла всегда локален для каждого потока

Правила разделения переменных (2)

Явное правило 1: Переменные, приведенные в `shared()`, являются глобальными для всех потоков

Явное правило 2: Переменные, приведенные в `private()`, являются локальными для каждого потока

Параллельные секции

```
#pragma omp parallel sections
// создан параллельный регион секций
{
#pragma omp section
    <код>
#pragma omp section
    <код>
}
// все секции выполняются одновременно в
// разных потоках
```

Параллельные секции

- Директива `sections` – распределение вычислений для отдельных фрагментов кода
- Фрагменты выделяются при помощи директивы `section`
- Каждый фрагмент выполняется однократно
- Разные фрагменты выполняются разными потоками
- Завершение директивы по умолчанию синхронизируется
- Директивы `section` должны использоваться только в статическом контексте

Редукции в циклах

```
int sum =0;
#pragma omp parallel for reduction (+ sum)
{
for (int i=0; i<1000; i++)
    sum = sum + func(i);
}
// reduction означает:
// -- создание приватной копии sum в
//    каждом потоке
// -- после завершения цикла сложение
//    значений всех приватных копий в sum
```


Редукции в циклах

Параметр `reduction` определяет список переменных, для которых выполняется операция редукции

- перед выполнением параллельной области для каждого потока создаются копии этих переменных,
- потоки формируют значения в своих локальных переменных
- при завершении параллельной области над всеми локальными значениями выполняются необходимые операции редукции, результаты которых запоминаются в исходных (глобальных) переменных

Барьеры

```
int sum =0;
#pragma omp parallel for reduction (+ sum)
{
for (int i=0; i<1000; i++)
    sum = sum + func(i);
}
// продолжение работы возможно только
// только при завершении всех потоков

// барьер == конец parallel for
// барьер == конец parallel sections
```

Барьеры и `nowait`

В некоторых случаях возникает потребность отключать синхронизацию,

Для этого существует оператор `nowait`

Это может быть выгодно, если за циклом следует второй цикл, данные в котором НЕ ЗАВИСЯТ от результатов первого цикла.

Барьеры и nowait

```
#pragma omp parallel
{
#pragma omp for nowait
    for(int i = 0; i < 5000; i++) {
        // ...
    }
#pragma omp for
    for(int j = 0; j < 1000; j++) {
        // ...
    }
#pragma omp barrier // здесь ждем завершения
    some_func(); // код после барьера
}
```

Барьеры и nowait

В рассмотренном примере потоки, которые освободились после обработки первого цикла, переходят к обработке второго цикла без ожидания остальных потоков.

В некоторых случаях это может повысить производительность, поскольку уменьшается время простоя потоков.

```
#pragma omp barrier
```

Создание барьера после nowait

Критические секции

С помощью критических разделов можно предотвратить одновременный доступ к одному сегменту кода из нескольких потоков. Один поток получает доступ только тогда, когда другие не обрабатывают данный код. Конструкция имеет следующий вид:

```
#pragma omp critical  
{  
    ...  
}
```

Критические секции

Применение критических секций там, где они не нужны, нежелательно :

- из-за критических секций потокам приходится ждать друг друга, а это уменьшает приращение производительности, достигнутое благодаря распараллеливанию кода,
- на вход в критические секции и на выход из них также затрачивается некоторое время.

Выполнение в одном потоке внутри параллельного фрагмента

```
#pragma omp parallel num_threads(4)
{ ... // что-то выполняем параллельно в 4 потоках
  #pragma omp single {
    int numTh = omp_get_num_threads();
    cout <<"\n"<<numThreads="
<<numTh<<"\n";
  }
} //Директива single означает, что соответствующая
секция должна быть выполнена одним потоком.
Этим потоком с равной вероятностью может
оказаться любой из существующих
```


Выполнение в одном потоке внутри параллельного фрагмента

```
Hello world from thread = 0  
Hello world from thread = 1
```

```
numThreads=4
```

- Потоков 4, но сообщение выдано только одним потоком !

Два базовых типа конструкций OpenMP

- Директивы `#pragma`
- Функции исполняющей среды OpenMP

Функции OpenMP служат в основном для изменения и получения параметров среды. Кроме того, OpenMP включает API-функции для поддержки некоторых типов синхронизации.

Прагмы синхронизации

`#pragma omp single` – исполняет следующую команду только с помощью одного (случайного) потока

`#pragma omp barrier` – удерживает потоки в этом месте, пока все потоки не дойдут до него

`#pragma omp atomic` – атомарно исполняет следующую операцию доступа к памяти (т.е. без прерывания от других ветвей)

`#pragma omp critical [имя потока]` – позволяет только одному потоку перейти к исполнению следующей команды

Критическая секция

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N; i++)
    PerformSomeComputation(i);
// внутри этой функции плохой код
```



ОШИБКА!

Функция *PerformSomeComputation* изменяет значение глобальной переменной — гонка данных

```
int global = 0;
void PerformSomeComputation(int i) {
    global += i;
}
```

Критическая секция

Избежать ситуацию возникновения гонки за ресурсами позволяет использование критических секций:

```
void PerformSomeComputation(int i) {  
    #pragma omp critical  
    {  
        global += i;  
    }  
}
```

Критическая секция

`#pragma omp critical` позволяет только одному потоку выполнить операцию даже внутри параллельного фрагмента

```
int a[MAXN], sum=0;
#pragma omp parallel for
for (int i = 0; i < N; i++)
{
    #pragma omp critical
    sum +- a[i];
}
```

Библиотека функций

```
void omp_set_num_threads(int numThreads)
```

Позволяет назначить максимальное число потоков для использования в следующей параллельной области (если это число разрешено менять динамически).

Вызывается из последовательной области программы

Библиотека функций

```
int omp_get_max_threads()
```

Возвращает максимальное число потоков

```
int omp_get_num_threads()
```

Возвращает фактическое число потоков в параллельной области программы

Библиотека функций

```
int omp_get_thread_num()
```

Возвращает номер потока

```
int omp_get_num_procs()
```

Возвращает число процессоров, доступных
приложению

Библиотека функций

```
int omp_in_parallel()
```

Возвращает true, если вызвана из параллельной области программы

Библиотека функций

Блокировки

```
omp_lock_t  myLock;  
omp_init_lock( &myLock );
```

```
omp_set_lock( &myLock );  
omp_unset_lock( &myLock );
```

```
omp_test_lock ( &myLock );
```

...

Правильный код блокировки

```
omp_lock_t  myLock;
omp_init_lock(&myLock);
#pragma omp parallel sections
{
    #pragma omp section
    {
        ...
        omp_set_lock(&myLock);    ...
        omp_unset_lock(&myLock); ...
    }
    #pragma omp section
    {
        ...
        omp_set_lock(&myLock);    ...
        omp_unset_lock(&myLock); ...
    }
}
```

Лабораторная работа № 3

Реализовать задание к работе № 2
средствами OpenMP