```
# Введение в OpenMP
```


Начнем с небольшого примера

Основа синтаксиса программной модели **OpenMP** - директивы препроцессора, или так называемые прагмы, например: `#pragma omp construct [clause[clause]]`

В дополнение к этому могут использоваться так же переменные окружения либо библиотечные функции, подключаемые через: `#include <omp.h>`

Большинство директив **OMP** применяются к блоку кода. Блок кода - структурная единица кода в ЯП, имеющая одну точку входа и одну точку выхода, состоящая из набора инструкций

```cpp
#include <iostream>

int main() {
    int ID = 0;
    std::cout << "Hello (" << ID << ") ";
    std::cout << "world (" << ID << ")\n";
    return 0;
}
```

```
Output:
Hello (0) world (0)
```

```cpp
#include <iostream>

int main() {
    #pragma omp parallel
    {
        int ID = 0;
        std::cout << "Hello (" << ID << ") ";
        std::cout << "world (" << ID << ")\n";
    }
    return 0;
}
```

```
Output:
Hello (Hello (0) world (0)
0) world (0)
```

В данном примере, при добавлении директивы, необходимо выделить ту часть кода, которая будет запускаться в параллель, в отдельный блок `{}` и перед ним поставить эту директиву. Тогда этот блок кода уже будет называться **регионом параллельного кода**. Можно заметить, что в выводе получается какая-то мешанина. Давайте попробуем ее отделить, и понять, что какому потоку пренадлежит:

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

int main() {
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        std::cout << "Hello (" << ID << ") ";
        std::cout << "world (" << ID << ")\n";
    }
    return 0;
}
```

```
Output:
Hello (Hello (0) world (0)
1) world (1)
```

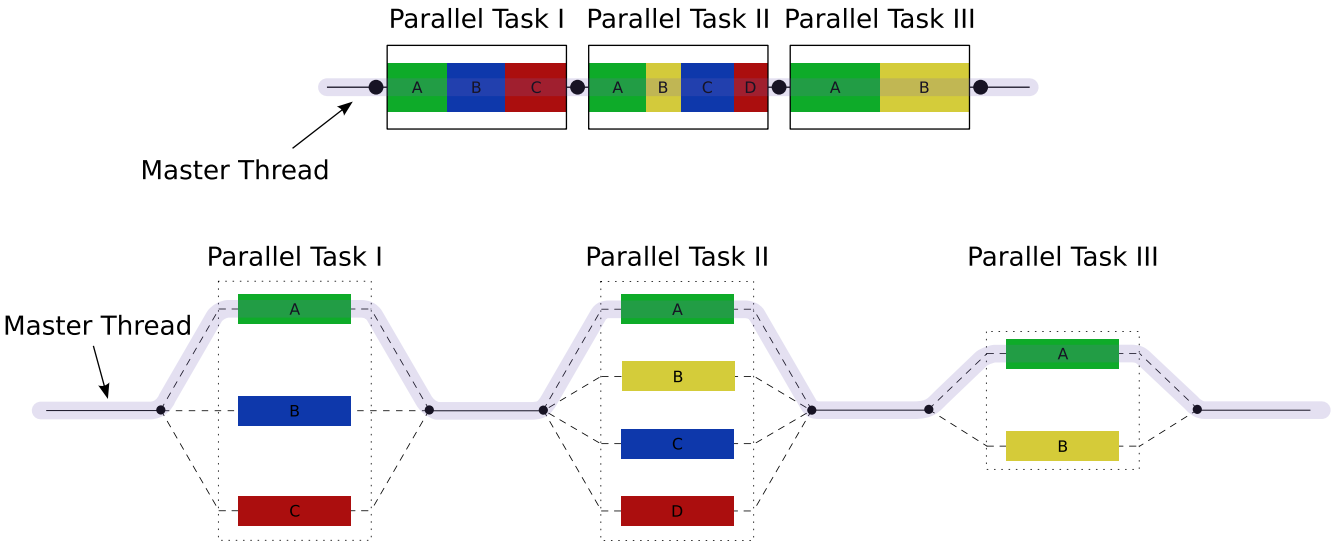
Для этого можно воспользоваться библиотечной функцией `omp_get_thread_num()`. Эта функция, при вызове внутри параллельного региона вернет номер идентификатора потока, в котором код в данный момент исполняется.

Праграммная модель **OMP** основана на использовании разделямой памяти. Таким образом, параллельные потоки могут коммуницировать через обмен данными, разделяемыми переменными. Однако необходимо помнить о важности контроля доступов к разделяемым данным из нескольких потоков, и что может возникнуть такая ситуация, как **гонка данных**.

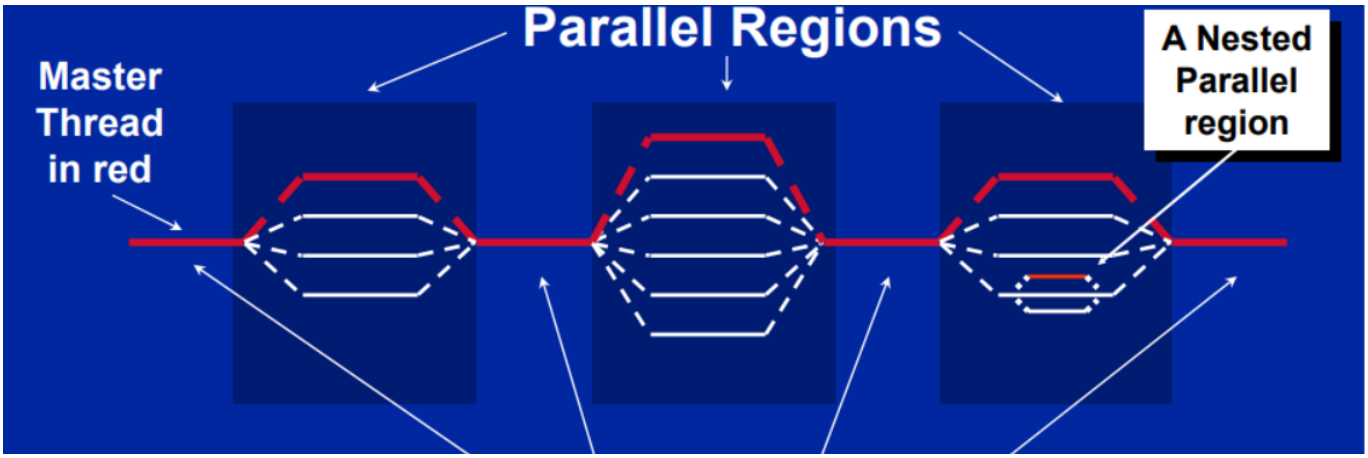
Так, например, в коде выше, из-за того, что потоковый ввод-вывод одновременно использовался разными потоками одновременно, возникла мешанина из вывода в консоль.

Попробовать самостоятельно решить проблему с выводом в консоль из разных потоков.

Создание потоков. Fork-join модель



На рисунке выше представлена модель параллеллизма fork-join, лежащая в основе **OMP**. Суть ее проста: есть главный поток (**Master Thread**), который отвечает за создание группы параллельных потоков, которые будут, совместно с главным потоком, решать задачи (**Parallel Task I, II и III**). Помимо этого, порожденные потоки внутри параллельного региона могут создавать еще группы потоков, таким образом, создавая уже модель вложенного параллелизма



Тогда программа будет представлять из себя чередующуюся последовательность из участков последовательного кода, однопоточные, и многопоточных участков.

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

int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        std::cout << "Hello (" << ID << ") ";
        std::cout << "world (" << ID << ")\n";
    }
    return 0;
}
```

В данном случае программа будет иметь следующую структуру: Сперва исполняется последовательный код, до момента начала параллельного региона, для которого создается набор параллельных потоков, которые будут исполнять свою копию кода (со своим уникальным значением ID), после чего, в конце региона, они сойдутся, и дальше, до конца работы программы, будет выполняться снова один поток.

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

int main() {
    const char* name = "world";
    #pragma omp parallel num_threads(4)
    {
        int ID = omp_get_thread_num();
        std::cout << "Hello (" << ID << ") ";
        std::cout << name <<" (" << ID << ")\n";
    }
    return 0;
}
```

Еще один способ задать количество параллельных потоков непосредстенно только для параллельного региона - использовать **num_threads(N) clause** для прагмы.

При этом, важно отметить, что переменная ID будет уникальна для каждого потока, тогда как name уже будет общей, разделяемой переменной для них.

Синхронизация

Для доступа к разделяемым переменным, в тех ситуациях, когда их значения обновляются и считываются из разных потоков, необходимо использовать примитивы синхронизации. Есть верхнеуровневые примитивы синхронизации и нихкоровневые. К верхнеуровневым можно отнести:

- Критические секции
- Атомарные операции
- Барьеры
- Упорядочивание доступов к данным

Критические секции

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

int main() {
    const char* name = "world";
    #pragma omp parallel num_threads(4)
    {
        int ID = omp_get_thread_num();
        #pragma omp critical
        {
```

```
            std::cout << "Hello (" << ID << ") ";
            std::cout << name <<" (" << ID << ")\n";
        }
    }
    return 0;
}
```

```
Output:
Hello (0) world (0)
Hello (3) world (3)
Hello (1) world (1)
Hello (2) world (2)
```

В данном примере, для того чтобы разграничить выводимые в консоль сообщения, как раз таки можно воспользоваться критической секцией, которая в каждый момент времени может выполняться строго одним потоком из параллельного региона. Для синхронизации все потоки встают на spin-lock и ожидают до тех пор, пока кто-то, кто его захватил, не закончит выполнение критической секции.

Атомарные операции

Однако в некоторых случаях вовсе нет необходимости в обеспечивании последовательного исполнения блока инструкций, а всего навсего нужно обеспечить последовательный доступ к одной единственной переменной. Тогда на помощь могут прийти атомарные операции:

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

int func() {
    int ret_val{};
    for (int i = 0; i < 10; ++i) {
        ++ret_val;
    }
    return ret_val;
}

int main() {
    int sum{};
    #pragma omp parallel num_threads(4)
    {
        int local_sum = func();
        #pragma omp atomic
        sum += local_sum;
    }

    std::cout << sum << std::endl;
    return 0;
}
```

Засчет поддержки атомарных операций на уровне ISA и микроархитектуры, простейшие операции, такие как суммирование, умножение, проверка значения, проверка с обновлением и прочее, могут быть выполнены эффективнее. Однако если пытаться применять директиву atomic к более сложным конструкциям, то сработает оно аналогично случаю с критической секцией.

Барьеры

Барьеры служат в качестве точки синхронизации для параллельного региона. Когда потоки в последовательности инструкций встречают барьер, то они обязаны дождаться, пока все остальные потоки так же не дойдут до него.

Ниже пример параллельной секции, в которой используется директива barrier. Так, дойдя до этого места, все потоки будут дожидаться остальных. Помимо этого, кроме явных барьеров, есть еще и неявные, которые автоматически создаются в:

- конце каждого блока `omp for`, то есть как только поток закончит обработку своей части итераций первого цикла, он будет дожидаться момента, пока все остальные потоки так же не завершат работу. Это необходимо для того, чтобы синхронизировать работу потоков и иметь корректные данные по завершении работы цикла для выполнения, скажем, операции редукции и пр.
- конце параллельного региона

Можно отключить неявную синхронизацию, используя модификатор `nowait`

```
#include <omp.h>

using namespace std;

int main() {

    int A[100], B[100], C[100], N = 100;
    int id, i;

    #pragma omp parallel shared (A, B, C) private(id, i)
    {
        id=omp_get_thread_num();
        A[id] = id;
        #pragma omp barrier
        #pragma omp for
        for(i=0;i<N;i++){C[i]+=A[i];}
        #pragma omp for nowait
        for(i=0;i<N;i++){ B[i]*=C[i]; }
        A[id] = id - 1;
    }
    return 0;
}
```

master construct

Для обозначения блока кода, в который зайдет только главный поток используется директива master:

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    { exchange_boundaries(); }
    #pragma omp barrier
    do_many_other_things();
}
```

При этом другие потоки неявно не ожидают завершения главного. Для этого необходимо использовать барьер.

single construct

В отличие от critical секции single блок уже имеет неявный барьер в конце.

Параллельные циклы

Создаваемые по умолчанию потоки, как это было в примерах ранее, выполняют копию одного и того же кода параллельного региона (**SPDM - Single Program Multiple Data**). Для какого-то разделения работы (**worksharing**) между потоками параллельного региона, вместо простого дублирования, можно воспользоваться библиотечными функциями для определения контекста исполнения кода параллельного региона. В качестве контекста здесь рассматривается, например, номер потока, исполняющего параллельный регион. Чтобы получить информацию о текущем потоке можно воспользоваться методом `omp_get_thread_num()`. Тогда, например, можно разделить M итераций исходного цикла на K частей между N потоками, где K = M/N.

Но более удобным в данном случае, когда есть цикл, итерации которого можно обрабатывать в параллель, будет подход с использованием `#pragma omp parallel for`

К используемой ранее прагме добавляется еще одно слово, указывающее на то, что итерации следующего за прагмой цикла можно запустить в параллель.

Естественно, итериций цикла может быть сильно больше имеющегося числа потоков, тогда они будут либо динамически, либо статически распределены между имеющимися потоками.

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

int main() {
    int sum{};
    #pragma omp parallel for
    for (int i = 0; i < 1000; ++i)
    {
        #pragma omp atomic
        sum += 1;
    }
```

```
    std::cout << sum << std::endl;
    return 0;
}
```

В данном случае переменная i будет локальной для каждого потока, а sum - разделяемая. Мотивацией для использования конструкции параллельного цикла может послужить следующий пример кода: Последовательная версия

```
for(i=0;I<N;i++) { a[i] = a[i] + b[i];}
```

Параллельная без **parallel for**

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1) iend = N;
    for(i=istart;I<iend;i++) { a[i] = a[i] + b[i];}
}
```

С использованием **parallel for**

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

Да, использование **omp for** конструкции можно как в примере ранее совместить с объявлением параллельного региона: **omp parallel for**, либо же вынести отдельно.

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

int main() {
    int sum{};
    int i;
    #pragma omp parallel for
    for (i = 0; i < 1000; ++i)
    {
        #pragma omp atomic
        sum += 1;
    }
```

```
    std::cout << sum << std::endl;
return 0;
}
```

Для итератора параллельного цикла, в примере выше - переменная i, создается локальная копия для каждого потока, то есть, для нее нет необходимости в атомарном доступе.

Задание: Посмотрите на пример выше, сравните две переменные: i и sum. Обе переменные меняются на каждой итерации в параллельном цикле. Однако для i атомарный доступ не используется, а для sum - да. Вопрос: как здесь можно избежать издержек на атомарный доступ для переменной sum?

Подсказка: В ответе можно воспользоваться модификаторами для прагмы, либо же вовсе обойтись без них.

Далее последует описание решения возникшей проблемы.

Редукция

Возникшая выше ситуация описывает стандартный паттерн в параллельных вычислениях, когда необходимо аккумулировать в какой-то переменной результат вычислений из параллельных потоков. И тогда между итерациями параллельного цикла возникает реальная зависимость по данным, которую можно разрешить с использованием редукции. Редукция позволяет разделить одну глобальную переменную на N копий, каждую для отдельного потока. Произвести над ними манипуляции так же локально по потокам, после чего, по завершении параллельного региона, собрать все зачения воедино. Такой подход применим, в частности, тогда, когда необходимо получить сумму или произведение каких-то чисел со всех потоков в одной переменной. Тогда оптимальным решением будет следующее:

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

int main() {
    int sum{};
    int i;
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < 1000; ++i)
    {
        sum += 1;
    }
    std::cout << sum << std::endl;
    return 0;
}
```

для reduction в качестве аргументов задается глобальная переменная, в которой необходимо собрать значения (локальные копии) со всех потоков, а так же операция, способ, которым эти значения будут объединены в формате (операция:переменная).

Подробнее про редукцию

Начальные значения локальных копий редуцируемой переменной будут задаваться в зависимости от выполняемой операции редукции (для + это 0, а для * - 1).

Operator	Initial value
+	0
*	1
-	0

C/C++ only	
Operator	**Initial value**
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	**Initial value**
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0

.IAND.	All bits on
.EQV.	.true.
MIN*	Largest pos. number
MAX*	Most neg. number

Разделяемые данные

Поскольку OpenMP использует модель параллелизма с разделяемой памятью, то ряд данных будет общим для создаваемых потоков:

- Глобальные переменные (область видимости внутри всего файла, статические переменные)
- Память, выделяемая динамически
- Переменные на стеке вне параллельных регионов так же будут разделяемыми

```
double A[10];
int main() {
    int index[10];
    #pragma omp parallel
    work(index);
    printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
    double temp[10];
    static int count;
    ...
}
```

В данном случае разделяемые переменные:

- A, index, **count**

Приватные переменные:

- temp

Для управления областью доступности внешних переменных используются следующие модификаторы:

- shared
- private

- firstprivate
- lastprivate

```
void wrong() {
    int tmp = 0;
    #pragma omp for private(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf(“%d\n”, tmp);
}
```

В целом, по названиям можно понять смысл использования этих модификаторов. Стоит только уточнить, чем firstprivate и lastprivate отличаются от private. По умолчанию, начальное значение private переменных внутри параллельного региона никак не определено по стандарту. Для того, чтобы гарантировать, что переменная будет проинициализирована значением внешней, исходной, переменной, нужно использовать firstprivate. lastprivate же позволяет обновить значение внешней переменной значением из локальной переменной, которое было получено на последней итерации параллельного цикла.