ЛЕКЦИЯ 2

Среда программирования OpenMP.

Синхронизация

1. Синхронизация в среде OpenMP

Как и в любой среде многопоточного программирования, в OpenMP важную роль играет синхронизация. Синхронизация необходима, если различные потоки работают с общими данными. Если чтение и запись или повторная запись общей переменной производятся без синхронизации, то результирующее значение переменной считается неопределенным.

Самым простым способом защитить данные от возможных проблем, связанных с одновременным доступом, является директива `atomic`. Эта директива применяется к оператору-выражению одного из следующих типов:

```
x op= expr;
x ++;
++ x;
x --;
-- x;
```

2. Пример использования директивы atomic

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int count = 0;
#pragma omp parallel
{
#pragma omp atomic
count++;
}
printf("Число потоков: %d\n", count);
}
```

3. Критические секции

Другим базовым механизмом синхронизации является механизм *критических секций*. Критическая секция в коде программы выделяется с помощью директивы `critical`, которая имеет следующий синтаксис:

```
#pragma omp critical [name]
```

Опция `name` является необязательной; если она отсутствует, то считается, что директива имеет зарезервированное не специфицированное имя. Таким образом, каждой критической секции ставится в соответствие некоторое имя. Синхронизация обеспечивается следующим образом: критические секции с одинаковыми именами не могут выполняться одновременно.

4. Пример использования директивы critical

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int n;
#pragma omp parallel
      {
#pragma omp critical
            {
n=omp_get_thread_num();
printf("Нить %d\n", n);
             }
        }
}
```

5. Распределение вычислений между потоками

Директива `parallel` позволяет инициировать параллельное выполнение участка программы группой потоков. Используя информацию о номере потока, которая доступна через функцию `omp_get_thread_num`, и механизмы синхронизации, можно разрабатывать достаточно сложные параллельные программы по технологии, аналогичной применяемой в POSIX Threads. При этом процесс разработки остается достаточно сложным и низкоуровневым.

6. Директива master

В некоторых случаях требуется ограничить набор потоков, выполняющих некоторый фрагмент кода. Это достигается при помощи директивы `master`, имеющей синтаксис:

```
#pragma omp master
```

Оператор, к которому применяется данная директива, выполняется только главным потоком.

В OpenMP предусмотрен также ряд других директив синхронизации, о которых более подробная информация содержится в [16].

7. Пример использования директивы master

```
#include <stdio.h>
int main(int argc, char *argv[])
{ int n;
#pragma omp parallel private(n)
            { n=1;
              #pragma omp master
                        { n=2;
                                }
             printf("Первое значение n: %d\n", n);
             #pragma omp barrier
             #pragma omp master
                          {n=3;
                            }
              printf("Второе значение n: %d\n", n);
              }
}
```

8. Директива single

Если в параллельной области какой-либо участок кода должен быть выполнен лишь один раз, то его нужно выделить директивами single:#pragma omp single [опция [[,] опция]...].
ОПЦИИ: private(список), firstprivate(список); copyprivate(список) – после выполнения нити, содержащей конструкцию single , новые значения переменных списка будут доступны всем одноименным частным переменным (private и firstprivate), описанным в начале параллельной области и спользуемым всеми её потоками (опция не может использоваться совместно с опцией nowait, переменные списка не должны быть перечислены в опциях private и firstprivate данной директивы single);
nowait – после выполнения выделенного участка происходит неявная барьерная синхронизация параллельных потоков (их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки), если в подобной задержке нет необходимости, опция nowait позволяет потокам, дошедшим до конца участка, продолжить выполнение без синхронизации с

9. Пример использования директивы single

```
#include <stdio.h>
int main(int argc, char *argv[])
{
#pragma omp parallel
        {
        printf("Сообщение 1\n");
        #pragma omp single nowait
                {
                printf("Одна нить\n");
                }
printf("Сообщение 2\n");
        }
}
```

10. Информационные зависимости

Одной из основных причин, которая не позволяет провести эффективное распараллеливание, является наличие информационных зависимостей между операторами программы. Информационные зависимости имеют место в случае, когда один из операторов использует результаты работы другого оператора. Например, считывает значения переменной, измененной другим оператором.

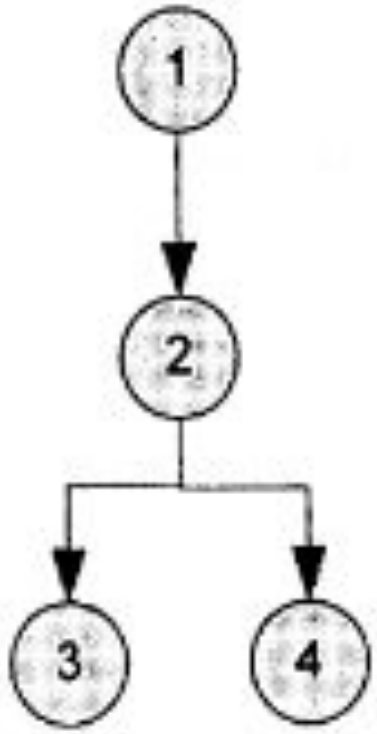
Будем говорить, что два оператора имеют *зависимость по данным*, если 1) оба обращаются к общему участку памяти, и 2) хотя бы одно из этих обращений является записью. Помимо зависимости по данным, существует также *зависимость по управлению* — когда один из операторов является оператором, влияющим на поток управления, а выполнение второго зависит от выполнения первого.

11. Примеры информационной зависимости

```
1: a ++;
2: f(a);
```

Оператор в строке 2 зависит от оператора в строке 1, так как в нем изменяется аргумент функции *f* — переменная *a*. Это — типичный пример зависимости по данным.

```
1: a ++;
2: if(a
   ==1) {
3: e ++;
4: b = c;
5: }
```



Операторы в строках 3 и 4 зависят от условного оператора в строке 2, который зависит от оператора в строке 1. Операторы в строках 3 и 4 независимы. Это — типичный пример зависимости по управлению.

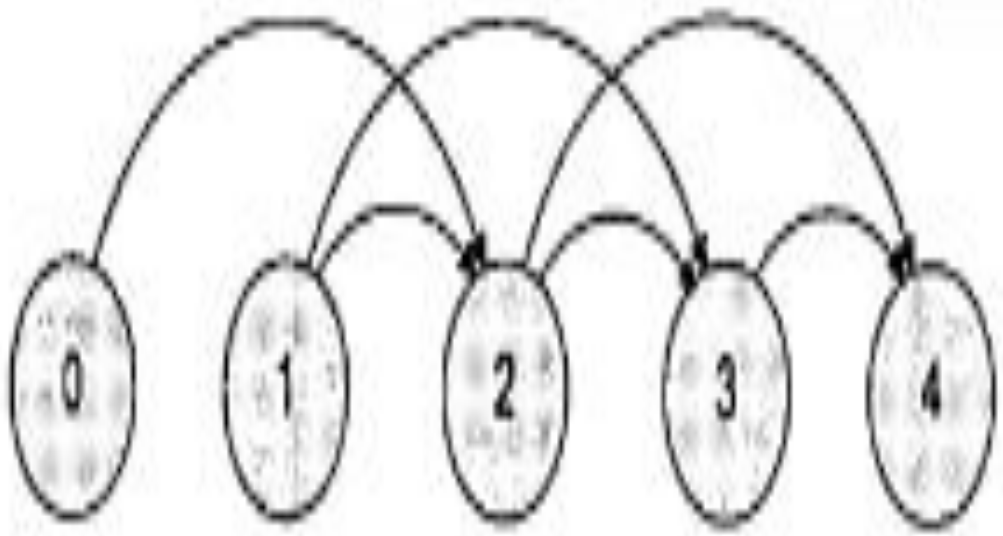
12. Информационная зависимость итераций

Важным частным случаем информационной зависимости является зависимость между итерациями цикла. Две итерации цикла зависят друг от друга, если 1) они обращаются к общим данным и 2) хотя бы одно из этих обращений является записью. Пример цикла с зависимостями между итерациями приведен на рис. 23. Для пяти итераций построен граф информационных зависимостей: итерация `j` зависит от итераций `j - 1` и `j - 2`.

Итерации цикла, между которыми существует информационная зависимость, необходимо выполнять в порядке, определяемом этой зависимостью. В частности, зависимые итерации не могут выполняться одновременно.

13. Пример информационной зависимости итераций

```
for(j = 0; j < n;
j ++)
    a[j] = a[j -
1] + a[j - 2];
```



14. Разбиение на независимые блоки. Директива section

Директива применяется к составному оператору, т.е. к последовательности операторов, заключенных в фигурные скобки. Тело составного оператора должно быть представлено в виде последовательности операторов (секций), каждому из которых, за исключением, быть может, первого, предшествует директива `section`, имеющая следующий синтаксис:

```
#pragma omp sections
{
#pragma omp section
        statement1
#pragma omp section
        statement2
...
}
```

15. Пример применения директивы sections

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{ int n;
      #pragma omp parallel private(n)
      { n=omp_get_thread_num();
        #pragma omp sections
                  {#pragma omp section
                        {printf("Первая секция, процесс %d\n", n);}
                   #pragma omp section
                         {printf("Вторая секция, процесс %d\n", n);}
                    #pragma omp section
                         {printf("Третья секция, процесс %d\n", n);}
                  }
          printf("Параллельная область, процесс %d\n", n);
       }
}
```

16. Параллельные циклы

```
#pragma omp for [опции … ]
for (init-expr; var rel b; incr-expr)
   тело_цикла
```

или

```
#pragma omp for [опция [[,] опция]...]

for (init-expr; var rel b; incr-expr)
   тело_цикла
```

17. Требования к оператору цикла

- for([целочисленный тип] i = инвариант цикла;
- i {<,>,=,<=,>=} инвариант цикла;
- i {+,-}= инвариант цикла)

- Эти требования введены для того, чтобы OpenMP мог при входе в цикл точно
- определить число итераций. Если директива параллельного выполнения стоит перед гнездом циклов, завершающихся одним оператором, то директива действует только на самый
- внешний цикл. Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно
- делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции lastprivate.

18. Возможные опции:

private(список) , firstprivate(список) , lastprivate(список) ;
reduction(оператор:список) – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждом потоке, локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги), над локальными копиями переменных после завершения всех итераций цикла выполняется заданный оператор; оператор это: для языка Си – + , * , - , & , | , ^ , && , || , порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;
schedule(type[, chunk]) – опция задаёт, каким образом итерации цикла распределяются между потоками;
collapse(n) — опция указывает, что n последовательных тесновложенных циклов ассоциируется с данной директивой, для циклов образуется общее пространство итераций, которое делится между потоками, если опция collapse не задана, то директива относится только к одному непосредственно следующему за ней циклу;
nowait – в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки, опция nowait позволяет потокам, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными

19. Использование директивы for

Последовательная функция:

```
1:  void vsum(int n, double* a, double* b,
      double* c)
2:  {
3:  int i;
4:  for(i = 0; i < n; i ++)
5:    c[i] = a[i] + b[i];
6:  }
```

Параллельная функция:

```
1:  void vsump(int n, double* a, double* b,
      double* c)
2:  {
3:    int i;
4:    #pragma omp for
5:    for(i = 0; i < n; i ++)
6:      c[i] = a[i] + b[i];
7:  }
```

20. Устранение зависимости по данным

```
1:  double dotprod(int n, double* a, double* b)
2:  {
3:    int i;
4:    double s;
5:    s = 0;
6:    for(i = 0; i < n; i ++)
7:      s += a[i] * b[i];
8:    return s;
9:  }
```

```
1:  double dotprods(int n, double* a, double* b)
2:  {
3:    int i;
4:    double s;
5:    s = 0;
6:    #pragma omp parallel for
7:    for(i = 0; i < n; i ++)
8:      #pragma omp atomic
9:      s += a[i] * b[i];
10:   return s;
11: }
```

21. Использование опции reduction

```
1:  double dotprodp(int n, double* a, double* b)
2:  {
3:    int i;
4:    double s;
5:    s = 0;
6:    #pragma omp parallel for reduction(+:s)
7:    for(i = 0; i < n; i ++)
8:      s += a[i] * b[i];
9:    return s;
10: }
```

22. Сравнение последовательного и параллельного варианта

Таблица Время работы последовательного и параллельного (с редукцией) вариантов функции вычисления скалярного произведения при различной длине векторов

Время работы варианта, сек	Число итераций				
	10000000	1000000	100000	10000	1000
	Длина вектора				
	1000	10000	100000	1000000	10000000
последователь-ного	13	13	12	12	13
параллельного	36	9	7	7	6