ЛЕКЦИЯ 3

Распределения итераций и дополнительные сведения о синхронизации

1. Пример параллельно исполняемых итераций

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int A[10], B[10], C[10], i, n;
/* Заполним исходные массивы */
   for (i=0; i<10; i++){ A[i]=i; B[i]=2*i; C[i]=0; }
#pragma omp parallel shared(A, B, C) private(i, n)
   {
      /* Получим номер текущей нити */
      n=omp_get_thread_num();
#pragma omp for
      for (i=0; i<10; i++)
      {
         C[i]=A[i]+B[i];
         printf("Нить %d сложила элементы с номером %d\n",
                n, i);
      }
   }
}
```

На каждой i -ой итерации суммируются i -ые элементы массивов A и B, а результат записывается в i -ый элемент массива C. Также на каждой итерации будет напечатан номер потока, выполнившего данную итерацию.

2. Распределение итераций по потокам

Итерации цикла, к которому применена директива `for`, могут по-разному распределяться по потокам. Повлиять на стратегию распределения, применяемую по умолчанию, позволяет опция `schedule` директивы `for`. Данная опция предусматривает два аргумента. Первый определяет способ распределения итераций. Второй необязательный аргумент задает число итераций в порции, которая служит единицей распределения нагрузки. Предусмотрены четыре различных варианта соответствующие различным комбинациям аргументам `schedule` (табл. 6). Если опция не указана, то применяется стратегия, установленная по умолчанию.

3. Опция schedule(type [, chunk]) задаёт, каким образом итерации цикла распределяются между нитями:

static – блочно-циклическое распределение итераций цикла; размер блока – chunk . Первый блок из chunk итераций выполняет нулевой поток, второй блок — следующий и т.д. до последнего потока, затем распределение снова начинается с нулевого потока. Если значение chunk не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между потоками.

dynamic – динамическое распределение итераций с фиксированным размером блока: сначала каждый поток получает chunk итераций (по умолчанию chunk=1), тот поток, который заканчивает выполнение своей порции итераций, получает первую свободную порцию из chunk итераций. Освободившиеся потоки получают новые порции итераций до тех пор, пока все порции не будут исчерпаны.

4. Опция schedule(type [, chunk]) - продолжение

guided – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины chunk (по умолчанию chunk=1) пропорционально количеству ещё не
распределённых итераций, делённому на количество потоков, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет лучше сбалансировать нагрузку потоков. Количество итераций в последней порции может оказаться меньше значения chunk.

auto – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр chunk при этом не задаётся.

runtime – способ распределения итераций выбирается во время работы
программы по значению переменной среды OMP_SCHEDULE . Параметр

5. Демонстрация влияния опции schedule на распределение итераций по потокам

```
int main(int argc, char *argv[])
{ int i;
#pragma omp parallel private(i)
        {
     #pragma omp for schedule (static)
    //#pragma omp for schedule (static, 1)
    //#pragma omp for schedule (static, 2)
     //#pragma omp for schedule (dynamic)
     //#pragma omp for schedule (dynamic, 2)
     //#pragma omp for schedule (guided)
     //#pragma omp for schedule (guided, 2)
       for (i=0; i<10; i++)  { printf("Нить %d выполнила
итерацию %d\n“, omp_get_thread_num(), i);  sleep(1);}
          }
}
```

6. Распределение итераций по потокам

i	static	static, 1	static, 2	dynamic	dynamic, 2	guided	guided, 2
0	0	0	0	0	0	0	0
1	0	1	0	1	0	2	0
2	0	2	1	2	1	1	1
3	1	3	1	3	1	3	1
4	1	0	2	1	2	1	2
5	1	1	2	3	2	2	2
6	2	2	3	2	3	3	3
7	2	3	3	0	3	0	3
8	3	0	0	1	3	0	0
9	3	1	0	0	3	3	0

7. Результаты демонстрации влияния опции schedule на распределение итераций по потокам

В ячейках таблицы указаны номера потоков, выполнявших соответствующую итерацию. Во всех случаях для выполнения параллельного цикла использовалось 4 потока. Для динамических способов распределения итераций (dynamic , guided) конкретное распределение между нитями может отличаться от запуска к запуску. В таблице видна разница между распределением итераций при использовании различных вариантов. К наибольшему дисбалансу привели варианты распределения (static, 2) , (dynamic, 2) и (guided, 2) . Во всех этих случаях одному из потоков достаётся на две итерации больше, чем остальным. В других случаях эта разница сглаживается.

8. Задачи (tasks)

Директива task применяется для выделения отдельной независимой задачи:

#pragma omp task [опция [[,] опция]...]

Текущий поток выделяет в качестве задачи ассоциированный с директивой блок операторов.

Задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией.

Для гарантированного завершения в точке вызова всех запущенных задач используется директива taskwait :

#pragma omp taskwait

Поток, выполнивший данную директиву, приостанавливается до тех пор, пока не будут завершены все ранее запущенные данным потоком независимые задачи.

9. Опции директивы task

private(список), firstprivate(список), shared(список);
if(условие) — порождение новой задачи только при выполнении некоторого условия, если условие выполняется, то задача будет выполнена текущем потоком и немедленно;
untied — опция означает, что в случае откладывания задача может быть продолжена любой нитью из числа выполняющих данную параллельную область, если данная опция не указана, то задача может быть продолжена только породившей её нитью;
default(private|firstprivate|shared|none) – всем переменным в задаче, которым явно не назначен класс, будет назначен класс private , firstprivate или shared соответственно; none означает, что всем переменным в задаче класс должен быть назначен явно; в языке Си задаются только варианты shared или none.

10. Директива ***ordered***

Директивы ordered определяют блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле: #pragma omp ordered. Блок операторов относится к самому внутреннему из объемлющих циклов, а в параллельном цикле должна быть задана опция ordered Поток, выполняющий первую итерацию цикла, выполняет операции данного блока. Поток, выполняющий любую следующую итерацию, должен сначала дождаться выполнения всех операций блока всеми потоками, выполняющими предыдущие
итерации. Может использоваться, например, для упорядочения вывода от параллельных нитей.

11. Директива ordered и опция ordered

```
int main(int argc, char *argv[])
{ int i, n;
#pragma omp parallel private (i, n)
        {
             n=omp_get_thread_num();
            #pragma omp for ordered
             for (i=0; i<5; i++)
             {printf("Нить %d, итерация %d\n", n, i);
            #pragma omp ordered
                                {
     printf("ordered: Нить %d, итерация %d\n", n, i);
                                }
                }
          }
}
```

12. Синхронизация с использованием механизма замков (locks)

В качестве замков используются общие целочисленные переменные (размер должен быть достаточным для хранения адреса). Данные переменные должны использоваться только как параметры примитивов синхронизации. Замок может находиться в одном из трёх состояний: неинициализированный, разблокированный или заблокированный. Разблокированный замок может быть захвачен некоторым потоком. При этом он переходит в аблокированное состояние. поток, захвативший замок, и только она может его освободить, после чего замок возвращается в разблокированное состояние. Есть два типа замков: простые замки и множественные замки. Множественный замок может многократно захватываться одной нитью перед его освобождением, в то время как простой замок может быть захвачен только однажды. Для множественного замка вводится понятие коэффициента захваченности (nesting count). Изначально он устанавливается в ноль, при каждом следующем захватывании увеличивается на единицу, а при каждом освобождении уменьшается на единицу. Множественный замок считается разблокированным, если его коэффициент захваченности равен нулю.

13. Инициализация замков

Для инициализации простого или множественного замка используются соответственно функции :

void omp_init_lock(omp_lock_t *lock);

void omp_init_nest_lock(omp_nest_lock_t *lock);

После выполнения функции замок переводится в разблокированное состояние. Для множественного замка коэффициент захваченности устанавливается в ноль.

Для переведения простого или множественного замка в неинициализированное состояние используются функции:

void omp_destroy_lock(omp_lock_t *lock);

void omp_destroy_nest_lock(omp_nest_lock_t *lock);

14. Захват замков

Для захвата замков используются функции:
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
Вызвавший эту функцию поток дожидается освобождения замка, а затем захватывает его. Замок при этом переводится в заблокированное состояние. Если множественный замок уже захвачен данным потоком, то он не блокируется, а коэффициент захваченности увеличивается на единицу.
Для освобождения замков используются функции:
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_lock_t *lock);
Вызов этой функции освобождает простой замок, если он был захвачен вызвавшем данную функцию потоком. Для множественного замка уменьшает на единицу коэффициент захваченности. Если коэффициент станет равен нулю, замок освобождается. Если после освобождения замка есть потоки, заблокированные при попытке захватить данный замок, замок будет сразу же захвачен одним
из ожидающих потоков.

15. Пример использования замков

```
#include <stdio.h>
#include <omp.h>
omp_lock_t lock;
int main(int argc, char *argv[])
{ int n; omp_init_lock(&lock);
#pragma omp parallel private (n)
                { n=omp_get_thread_num();
                 omp_set_lock(&lock);
                 printf("Начало закрытой секции, нить %d\n", n);
                sleep(5);
                printf("Конец закрытой секции, нить %d\n", n);
                omp_unset_lock(&lock);
                  }
omp_destroy_lock(&lock);
}
```

16. Неблокирующие попытки захвата

Для неблокирующей попытки захвата замка используются функции:
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t *lock);
Данная функция реализует попытку захватить указанный замок. Если это удалось, то для простого замка функция возвращает 1, а для множественного замка – новый коэффициент захваченности. Если замок захватить не удалось, то в обоих случаях возвращается 0.

17. Пример использования функции omp_test_lock()

```
int main(int argc, char *argv[])
{ omp_lock_t lock; int n;
omp_init_lock(&lock);
#pragma omp parallel private (n)
        {
        n=omp_get_thread_num();
        while (!omp_test_lock (&lock))
              {printf("Секция закрыта, нить %d\n", n);
              sleep(2);
              }
        printf("Начало закрытой секции, нить %d\n", n);
        sleep(5);
        printf("Конец закрытой секции, нить %d\n", n);
        omp_unset_lock(&lock);
        }
omp_destroy_lock(&lock);
}
```

18. Директива flush

Поскольку в современных параллельных вычислительных системах может использоваться сложная структура и иерархия памяти, пользователь должен иметь гарантии того, что в необходимые ему моменты времени все потоки будут видеть единый согласованный образ памяти. Именно для этих целей предназначена директива flush: #pragma omp flush [(список)]. Выполнение данной директивы предполагает, что значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущего потока, будут занесены в основную память; все изменения переменных во время работы потока, станут видимы остальным потокам; если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. При этом операция производится только с данными вызвавшего потока, данные, изменявшиеся другими потоками, не затрагиваются. Поскольку выполнение данной директивы в полном объёме может повлечь значительных накладных расходов, а в данный момент нужна гарантия согласованного представления не всех, а лишь отдельных переменных, то эти переменные можно явно перечислить в списке директивы. До полного завершения операции никакие действия с перечисленными в ней переменными не могут

19. Неявное применение директивы flush

Неявно flush без параметров присутствует в директиве barrier , на входе и выходе областей действия директив parallel , critical , ordered , на выходе областей распределения работ, если не используется опция nowait , в вызовах функций omp_set_lock() , omp_unset_lock() , omp_test_lock() , omp_set_nest_lock() , omp_unset_nest_lock() , omp_test_nest_lock() , если при этом замок устанавливается или снимается, а также перед порождением и после завершения любой задачи (task). Кроме того, flush вызывается для переменной, участвующей в операции, ассоциированной с директивой atomic . Заметим, что flush не применяется на входе области распределения работ, а также на входе и выходе области действия директивы master.

20. Использование технологии OpenMP

1. Если целевая вычислительная платформа является многопроцессорной и/или многоядерной, то для повышения быстродействия программы чаще всего разумно порождать по одному потоку на вычислительное ядро, хотя это не является обязательным требованием.
2. Для первоначальной отладки может быть вполне достаточно одноядерного процессора, на котором порождается несколько потоков, работающих в режиме разделения времени.
3. Порождение и уничтожение потоков в OpenMP являются относительно недорогими операциями, однако многократное совершение этих действий (например, в цикле) может повлечь существенное увеличение времени работы программы.
4. Для того чтобы получить параллельную версию, сначала необходимо определить ресурс параллелизма программы, то есть, найти в ней участки, которые могут выполняться независимо разными потоками. Если таких участков относительно немного, то для распараллеливания чаще всего используются конструкции, задающие статический параллелизм, например, параллельные секции или низкоуровневое распараллеливание по номеру потока.

21. Синхронизация и ускорение OpenMP-программ

1. Наибольший ресурс параллелизма сосредоточен в циклах, поэтому наиболее распространенным способом распараллеливания является распределение итераций циклов. Если между итерациями некоторого цикла нет информационных зависимостей, то их можно раздать разным потокам для одновременного исполнения. Различные способы распределения итераций позволяют добиваться максимально равномерной загрузки потоков .
2. Статический способ распределения итераций позволяет уже в момент написания программы точно определить, какому потоку достанутся какие итерации. Однако он не учитывает текущей загруженности процессоров, соотношения времён выполнения различных итераций и некоторых других факторов.
3. Эти факторы в той или иной степени учитываются динамическими способами распределения итераций. Кроме того, возможно отложить решение по способу распределения итераций на время выполнения программы (например, выбирать его, исходя из текущей загруженности нитей) или возложить выбор распределения на компилятор и/или систему выполнения.
4. Обмен данными в OpenMP происходит через общие переменные. Это приводит к необходимости разграничения одновременного доступа разных потоков к общим данным. Для этого предусмотрены достаточно развитые средства синхронизации. При этом нужно учитывать, что использование излишних синхронизаций может существенно замедлить программу.

22. Особенности использование технологии OpenMP

1. Взяв за основу последовательный код, пользователь шаг за шагом может добавлять новые директивы, описывающие параллельные области. Нет необходимости сразу распараллеливать всю программу, её создание ведется последовательно, что упрощает и процесс программирования, и отладку.
2. Нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова функций OpenMP могут быть подставлены специальные «заглушки» (stubs), текст которых приведен в описании стандарта. Они гарантируют корректную работу программы в последовательном режиме– нужно только перекомпилировать программу и подключить другую библиотеку.
3. Технология OpenMP может использоваться совместно с другими технологиями параллельного программирования, например, с MPI. Обычно в этом случае MPI используется для распределения работы между несколькими вычислительными узлами, а OpenMP затем используется для распараллеливания вычислений на одном узле.

23. Фрагмент программы вычисления числа Пи.

```
//Для распараллеливания достаточно добавить в последовательную
программу //всего две строчки.
double f(double y) {return(4.0/(1.0+y*y));}
int main()
{ double w, x, sum, pi; int i;
int n = 1000000; w = 1.0/n;
sum = 0.0;
#pragma omp parallel for private(x) shared(w) reduction(+:sum)
            for(i=0; i < n; i++)
            {
             x = w*(i-0.5);
             sum = sum + f(x);
               }
pi = w*sum;
printf("pi = %f\n", pi);
}
```

24. Программа перемножения матриц

```
#include <stdio.h>
#include <omp.h>
#define N 4096
double a[N][N], b[N][N], c[N][N];
int main()
{ int i, j, k; double t1, t2;
// инициализация матриц
for (i=0; i<N; i++) for (j=0; j<N; j++) a[i][j]=b[i][j]=i*j;
t1=omp_get_wtime();
// основной вычислительный блок
#pragma omp parallel for shared(a, b, c) private(i, j, k)
        for(i=0; i<N; i++){
                for(j=0; j<N; j++){ c[i][j] = 0.0;
                        for(k=0; k<N; k++) c[i][j]+=a[i][k]*b[k][j];}
                }
t2=omp_get_wtime(); printf("Time=%lf\n", t2-t1);
}
```

25. Времена выполнения произведения матриц на узле суперкомпьютера СКИФ МГУ «ЧЕБЫШЁВ».

Ниже приведена зависимость времени выполнения перемножения матриц на узле суперкомпьютера СКИФ МГУ «ЧЕБЫШЁВ» [9] при изменении числа использованных потоков от 1 до 8. Использовался компилятор Intel 11.0 без дополнительных опций оптимизации, кроме –openmp

1	2	4	8
165.442016	114.413227	68.271149	39.039399

26. Литература

1. OpenMP Architecture Review Board (http://www.openmp.org/).
2. The Community of OpenMP Users, Researchers, Tool Developers and Providers (http://www.compunity.org/).
3. OpenMP Application Program Interface Version 3.0 May 2008 (http://www.openmp.org/mp-documents/spec30.pdf).
4. Что такое OpenMP? (http://parallel.ru/tech/tech_dev/openmp.html).
5. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. - 608 с.
6. Barbara Chapman, Gabriele Jost, Ruud van der Pas. Using OpenMP: portable shared memory parallel programming (Scientific and Engineering Computation). Cambridge, Massachusetts: The MIT Press., 2008. - 353 pp.
7. Антонов А.С. Введение в параллельные вычисления. Методическое пособие. -М.: Изд-во Физического факультета МГУ, 2002. - 70 с.
8. Антонов А.С. Параллельное программирование с использованием технологии MPI: Учебное пособие. -М.: Изд-во МГУ, 2004. - 71 с.
9. Суперкомпьютерный комплекс Московского университета (http://parallel.ru/cluster/).
10. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.
11. Лупин С.А., Посыпкин М. А. Технологии параллельного программирования. – М.: ИД «ФОРУМ»: ИНФРА-М, 2011. -208 с.