Введение в параллельные вычисления

д.т.н. Мокрова Наталия Владиславовна

асп. Морунов Егор, Сырко Денис

пятница	ауд. 118, 119	
12:50 – 14:20	Лекция	
	1 неделя	2 неделя
14:35 - 17:55	Лаб – КС44	Лаб - КС40

Выбор модели программирования

Системы с общей памятью

- C++11 threads (pthreads)
- OpenMP

Системы с распределенной памятью

- MPI = message passing interface

Гетерогенные системы (графические процессоры)

- CUDA

Лекция 9.
Использование технологии OpenMP

Распределение итераций циклов.

Считаем Пи.

Считаем интеграл.

The OpenMP® API specification for parallel programming http://openmp.org

Антонов А.С. Параллельное программирование с использованием технологии OpenMP. М.: Издво МГУ, 2009.

Примеры из учебника «Технологии параллельного программирования MPI и OpenMP» http://parallel.ru/tech/tech_dev/MPI%26OpenMP/examples

Примеры http://pro-prof.com/archives/1150
http://habrahabr.ru/post/147796/
http://pro-prof.com/archives/1150

Демонстрация примеров http://coliru.stacked-crooked.com/

Дополнительные переменные среды и функции

Переменная OMP_MAX_ACTIVE_LEVELS задаёт максимально допустимое количество вложенных параллельных областей.

Значение может быть установлено при помощи вызова функции omp_set_max_active_levels().

void omp_set_max_active_levels(int max);

Если значение max превышает максимально допустимое в системе, будет установлено максимально допустимое в системе значение. При вызове из параллельной области результат выполнения зависит от реализации.

Значение переменной OMP_MAX_ACTIVE_LEVELS может быть получено при помощи вызова функции.

int omp_get_max_active_levels(void);

Функция omp_get_level() выдаёт для вызвавшей нити количество вложенных параллельных областей в данном месте кода.

int omp_get_level(void);

При вызове из последовательной области функция возвращает значение 0.

Дополнительные переменные среды и функции

Функция omp_get_ancestor_thread_num() возвращает для уровня вложенности параллельных областей, заданного параметром level, номер нити, породившей данную нить.

int omp_get_ancestor_thread_num(int level);

Если level меньше нуля или больше текущего уровня вложенности, возвращается – 1.

Если level = 0, функция вернёт 0, а если level = omp_get_level(), вызов эквивалентен вызову функции omp_get_thread_num().

Функция omp_get_team_size() возвращает для заданного параметром level уровня вложенности параллельных областей количество нитей, порождённых одной родительской нитью.

int omp_get_team_size(int level);

Вызов эквивалентен вызову функции omp_get_num_threads().

Функция omp_get_active_level() возвращает для вызвавшей нити количество вложенных параллельных областей, обрабатываемых более чем одной нитью, в данном месте кода.

int omp_get_active_level(void);

При вызове из последовательной области возвращает значение 0.

Дополнительные переменные среды и функции

Переменная среды OMP_STACKSIZE задаёт размер стека для создаваемых из программы нитей. Значение переменной может задаваться в виде size | sizeB | sizeK | sizeM | sizeG, где size – положительное целое число, а буквы B, K, M, G (если не указано, размер задаётся в килобайтах). Если задан неправильный формат или невозможно выделить запрошенный размер стека, результат будет зависеть от реализации.

Например, в Linux в командной оболочке bash задать размер стека можно при помощи команды:

export OMP_STACKSIZE=2000K

Переменная среды OMP_WAIT_POLICY задаёт поведение ждущих процессов.

Если задано значение ACTIVE, то ждущему процессу будут выделяться циклы процессорного времени, а при значении PASSIVE ждущий процесс может быть отправлен в спящий режим, при этом процессор может быть назначен другим процессам.

Переменная среды OMP_THREAD_LIMIT задаёт максимальное число нитей, допустимых в программе.

Если значение переменной не является положительным целым числом или превышает максимально допустимое в системе число процессов, поведение программы будет зависеть от реализации.

Значение переменной может быть получено при помощи процедуры omp_get_thread_limit().

int omp_get_thread_limit(void).

Еще раз о хранении данных

- Значение приватных переменных не определено при входе и выходе из параллельного региона.
- Значение исходной переменной (до параллельного региона) не определено после выхода из региона!
- Приватная переменная никак не связана с глобальной переменной с тем же именем.
- Используйте first/last private условия для изменения такого поведения.

Еще раз об эффективности

Задача: извлечь корень из каждого элемента массива и поместить результат в другой массив.

```
#pragma omp parallel
{
        for (ptrdiff_t i = 0; i < n; i++)
                dst[i] = sqrt(src[i]);
}
```

Каждый поток вычисляет корень из всех элементов массива!

Как извлечь пользу из параллельности?

```
#pragma omp parallel
{
        #pragma omp for
        for (ptrdiff_t i = 0; i < n; i++)
                dst[i] = sqrt(src[i]);
}
```

Каждый создаваемый поток обрабатывает только отданную ему часть массива!

Сокращенная запись, комбинация нескольких директив в одну управляющую строку.

```
#pragma omp parallel for
        for (ptrdiff_t i = 0; i < n; i++) dst[i] = sqrt(src[i]);
```

Еще раз о циклах

```
c[ 0 ] = 1
c[ 1 ] = 2
c[ 2 ] = 3
c[ 3 ] = 4
c[ 4 ] = 5
c[ 5 ] = 6
c[ 6 ] = 7
c[ 7 ] = 8
c[ 8 ] = 9
c[ 9 ] = 10
```

```
5  #define N 10
6  int main()
7  {
8     int a [N], b [N], c [N];
9     int i, n;
10    for(i = 0; i < N; i++)
11     {
12          a[i] = 1; b[i] = i;
13     }
14 #pragma omp parallel shared (a, b, c) private (i,n) num_threads(5)
15     {
16         n = omp_get_thread_num();
17     #pragma omp for schedule (static) ordered
18         for (i = 0; i < N; i++)
19         {
20             c[i] = a[i] + b[i];
21         #pragma omp ordered
22          {
23             cout << "c[ " << i << " ] = "<< c[i] <<endl;
24          }
25         }
26     }
27 }
```

Итерации в произвольном порядке

Итерации – по возрастанию

Распределение итераций цикла

Параметры опции schedule

- **static** – блочно-циклическое распределение; итерации распределяются блоками по chunk итераций на нить; если нити закончились, а итерации еще есть, то распределение продолжается с первой нити; если число chunk не указано, итерации делятся поровну между нитями.
- **dynamic** динамическое распределение; сначала каждая нить получает chunk итераций; затем следующую порцию по chunk итераций получает первая освободившаяся нить.
- **guided** – динамическое распределение; изменяется размер порции: он уменьшается с начального значения до chunk, пропорционально количеству еще не распределенных итераций, деленному на количество нитей.
- **auto** – способ распределения итераций определяется компилятором и /или средой выполнения;
- **runtime** – способ распределения итераций выбирается во время работы программы по значению переменной среды OMP_SCHEDULE

Пример schedule

```
int main()
{
    srand(time(NULL));
    int A[20],B[20],C[20],i,n;
    for (i = 0; i < 20; i++)
        { A[i]=i; B[i]=2*i; C[i]=0; }
#pragma omp parallel shared (A, B, C) private (i, n) num_threads(5)
    {
    n = omp_get_thread_num();
    #pragma omp for schedule (static, 1)
        for (i = 0; i < 20;i++)
        {
            C[i] = A[i] + B[i];
            cout << "Thread " << n << " iteration: " << i << endl;
            sleep(rand()%1);
        }
    }
return 0;
}
```

```
Thread 1 iteration: 1
Thread Thread 1 iteration: 6
Thread 1 iteration: 11
0Thread 1 iteration: 16
 iteration: 0
Thread 0 iteration: 5
Thread 0 iteration: 10
Thread 0 iteration: 15
Thread 2 iteration: 2
Thread 2 iteration: 7
Thread 2 iteration: 12
Thread 2 iteration: 17
Thread 3 iteration: 3
Thread 3 iteration: 8
Thread 3 iteration: 13
Thread 3 iteration: 18
Thread 4 iteration: 4
Thread 4 iteration: 9
Thread 4 iteration: 14
Thread 4 iteration: 19
```

Распределение итераций цикла по нитям

Из книги Антонова

i	static	static,1	static,2	dynamic	dynamic,2	guided	guided,2
0	0	0	0	1	0	0	0
1	0	1	0	2	0	0	0
2	0	2	1	2	1	0	0
3	0	3	1	3	1	0	0
4	1	4	2	0	2	1	1
5	1	0	2	4	2	1	1
6	1	1	3	4	3	1	1
7	1	2	3	3	3	1	1
8	2	3	4	1	4	2	2
9	2	4	4	2	4	2	2
10	2	0	0	0	1	2	2
11	2	1	0	1	1	3	3
12	3	2	1	4	2	3	3
13	3	3	1	3	2	4	4
14	3	4	2	2	0	4	4
15	3	0	2	4	0	3	3
16	4	1	3	0	3	3	3
17	4	2	3	2	3	4	4
18	4	3	4	2	4	2	4
19	4	4	4	3	4	1	4

Считаем Пи параллельно

Для распараллеливания достаточно добавить в последовательную программу строку.

```
#pragma omp parallel for private (x), reduction (+:sum)
    for (int i = 0; i < numSteps; i++)
    {
        x = (i + .5)*step;
        sum = sum + 4.0/(1.+ x*x);
    }
    pi = sum*step;
```

```
The value of PI is 3.14159 Error is 2.06057e-13
The time to calculate PI was 0.0416173 seconds

The value of PI is 3.14159 Error is 2.26485e-14
The time to PARALLEL calculate PI was  0.027838 seconds
```

Пи

```
int main(int argc, char** argv)
{
    const unsigned long numSteps=5000000;
    double step; double pi=0; double sum=0.0; double x;
    double PI25DT = 3.14159265358979323846264643;
    step = 1./static_cast<double>(numSteps);
    double start = omp_get_wtime();
    for (int i = 0; i < numSteps; i++)
    {
        x = (i + .5)*step; sum = sum + 4.0/(1.+ x*x);
    }
    pi = sum*step;  sum=0.0;
    double stop = omp_get_wtime();
    cout << "The value of PI is " << pi << " Error is " << fabs(pi - PI25DT) << endl;
    cout << "The time to calculate PI was " ;
    double time = (stop - start); cout << time << " seconds\n" << endl;
     start = omp_get_wtime();
    #pragma omp parallel for private (x), reduction (+:sum)
    for (int i = 0; i < numSteps; i++)
    {
        x = (i + .5)*step;
        sum = sum + 4.0/(1.+ x*x);
    }
    pi = sum*step;
     stop = omp_get_wtime();
    cout << "The value of PI is " << pi << " Error is " << fabs(pi - PI25DT) << endl;
    cout << "The time to PARALLEL calculate PI was  " ;
     time = (stop - start);
    cout << time << " seconds\n" << endl;
    return 0;
}
```

Вычисление интеграла методом прямоугольников

листинг 3 метод левых прямоугольников. Параллельная реализация с OpenMP

```
01  float rect_integral(const std::function<float(float)> &fun,
02                      const float a, const float b, const int n) {
03    float h = fabs((b - a) / n);
04
05    float sum = 0;
06  #pragma omp parallel reduction (+: sum)
07  {
08  # pragma omp for
09    for (int i = 0; i < n; ++i)
10      sum += fun(a + i * h) * h;
11  } // pragma omp parallel
12
13    return sum;
14  }
```

http://pro-prof.com/archives/1150

Задано количество прямоугольников

Вычислить: $\int_{-1}^{1} x^2 = \frac{2}{3}$

В качестве аргумента программа принимает количество прямоугольников.

Чем больше прямоугольников на ограниченном интервале – тем меньше каждый прямоугольник, и точность должна расти.

Точность действительно повышается при увеличении количества прямоугольников с 10 до 100.

Однако, при очень большом их количестве точность может понизится.

OpenMP тут не причем (в примере при компиляции не использовался ключ *-fopenmp*).

В приведенном примере точность падает потому, что очень маленькое значение шага (*h*) умножается на очень большое значение счетчика (*i*). В младших разрядах шага неизбежно находится мусор, который обрабатывается. Мы наблюдаем ошибку, которую не видит компьютер, программа в этом случае не генерирует исключений.

Особенно трудно определить причину таких ошибок в параллельных программах.

Деградация точности при большом количестве прямоугольников

http://pro-prof.com/archives/1150/omp_2

```
rrrfer@linux-2oyq:~/project/OMP> g++ main.cpp -o main -std=c++11
rrrfer@linux-2oyq:~/project/OMP> ./main 10
0.6800000668
rrrfer@linux-2oyq:~/project/OMP> ./main 100
0.6667999029
rrrfer@linux-2oyq:~/project/OMP> ./main 200000000
0.2500000000
rrrfer@linux-2oyq:~/project/OMP>
```


Еще о точности

На точность может влиять количество работающих потоков и порядок вычисления.

В задании вычислить сумму ряда $\frac{1}{x^2}$ сначала при изменении *x* от 1 до 100000000, затем в обратном порядке.

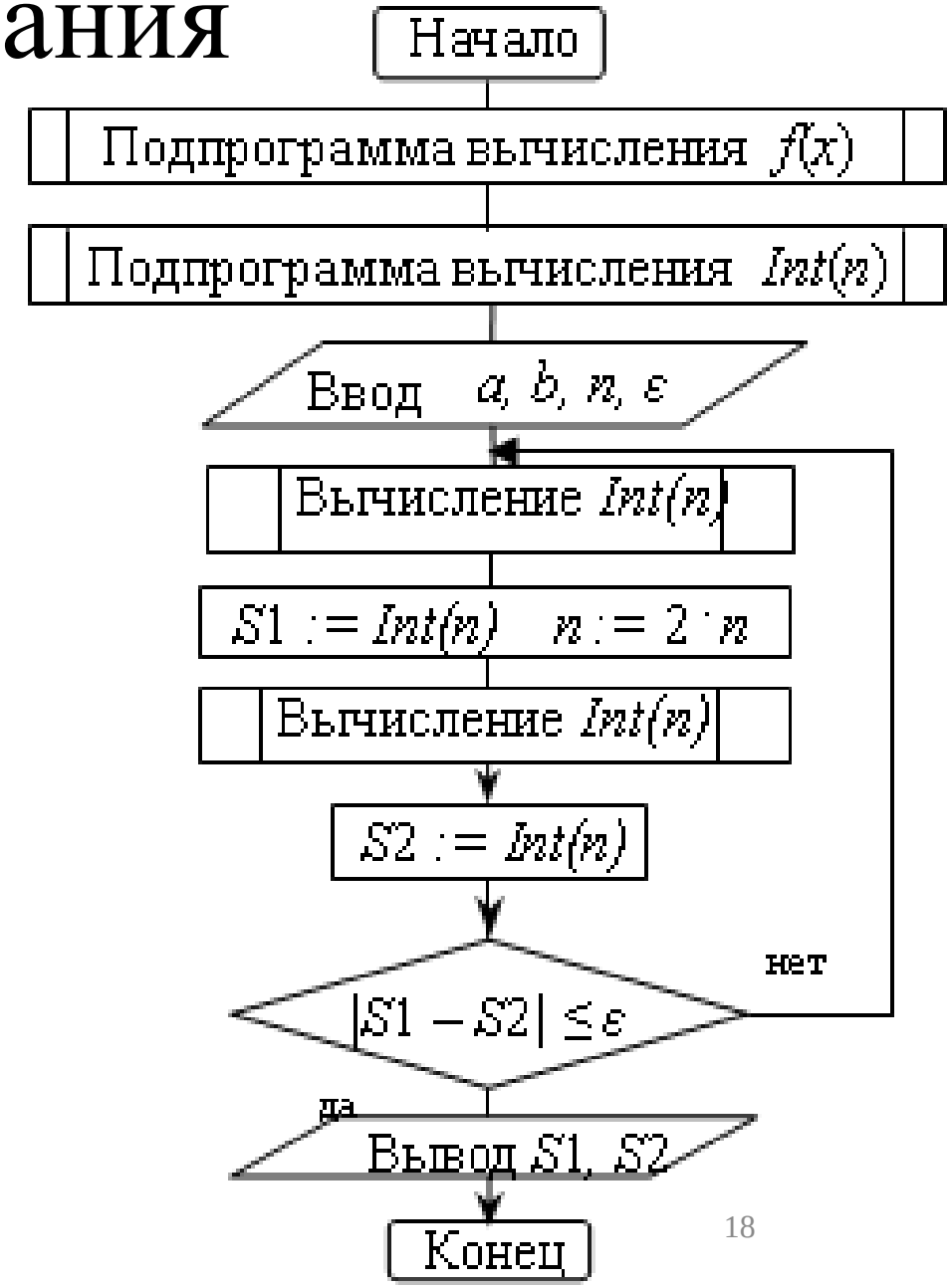
Результаты вычислений могут отличаться, вычисление в обратном порядке дает другой результат.

OpenMP не гарантирует определенный порядок вычислений, поэтому и может появляться неожиданная погрешность.

Задана точность интегрирования

Как распараллелить последовательную программу, если задана точность, а не количество прямоугольников?

Программа должна постепенно дробить шаг и считать сумму площадей прямоугольников до тех пор, пока разница суммарной площади на текущем шаге и предыдущем не окажется меньше точности.



Вычисление интеграла методом двойного пересчета

$|S(n) - S(2n)| \leq \varepsilon$ правило Рунге

Как использовать OpenMP

Определить количество итераций невозможно, т.к. интегрируемая функция может быть любой.

В OpenMP параллельный цикл должен иметь целочисленный счетчик.

> «Если количество итераций нельзя определить, то это плохой код, независимо от того, будет он работать последовательно или параллельно» (статья «Библиотека OpenMP. Параллельный цикл»).

Для использования OpenMP в этом случае достаточно определить заранее количество итераций внутреннего цикла и использовать в нем целочисленный счетчик.

Хоть OpenMP и задуман как средство постепенного распараллеливания последовательных программ, но программа все равно должна писаться определенным образом.

Метод левых прямоугольников с точностью

```
float rect_integral(const function<float(float)> &fun, const float a, const float b, const float eps)
{
    int n = 1; // в начале считаем только один прямоугольник
    float sum = fabs(b - a) * fun(a);
    float newSum;
    while (true)
    {
        n *= 2; // дробление шага
        float h = fabs(b - a) / n; newSum = 0;
        double start = omp_get_wtime();
        #pragma omp parallel shared (h) reduction (+: newSum)
        {
            # pragma omp for
                for (int i = 0; i < n; ++i)
                newSum += fun(a + i * h) * h;
        } // pragma omp parallel
        double end = omp_get_wtime();
    if (fabs(sum - newSum) < eps) // проверка точности
        { cout << " Значение n = " << n << endl; cout << " Time " << end - start << endl; break;}
    sum = newSum;
    }
    return sum;
}
```

Пример интеграл

```
float f(float x) {
    return x * x;
}
int main(int argc, char *argv[])
{
    float integral = rect_integral(f, -1, 1, 0.0001);
    cout << " Integral with eps = " << integral << "    Error " << fabs( 2/3.0 - integral) << endl;
}
```

```
Значение n = 1024
Time 5.3162e-05
Integral with eps = 0.666672    Error 5.08626e-06
```

Результаты:

Порядок точности 10^{-5} – 10^{-6}. Время последовательной и параллельной версии сравнимы. При увеличении точности время расчета последовательной версии растет значительно.

Нужно ли увеличивать точность?

И снова циклы

Реализация программы перемножения двух квадратных матриц.

В программе замеряется время на основной вычислительный блок, не включающий начальную инициализацию.

```
#define N 4096
double a[N][N], b[N][N], c[N][N];
int main()
{
    int i, j, k; double t1, t2;
    for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        a[i][j] = b[i][j] = i*j; // инициализация матриц
    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();
    cout << " Time = " << t2 - t1;
}
```

О времени вычислений

В таблице приведены времена выполнения примера на узле суперкомпьютера СКИФ МГУ «ЧЕБЫШЁВ».

Использовался компилятор Intel 11.0 без дополнительных опций оптимизации, кроме –fopenmp.

Количество нитей	1	2	4	8
Си	165.442016	114.413227	68.271149	39.039399
Фортран	164.433444	115.100835	67.953780	39.606582

Время выполнения произведения матриц на узле суперкомпьютера СКИФ МГУ «ЧЕБЫШЁВ»

(из книги Антонова).

Еще раз о преимуществах OpenMP

- За счет идеи **"инкрементального распараллеливания"** OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы.
- При этом, OpenMP – достаточно **гибкий механизм**, предоставляющий разработчику большие возможности контроля над поведением параллельного приложения.
- Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована **в качестве последовательной** программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова процедур OpenMP могут быть подставлены заглушки (stubs), текст которых приведен в спецификациях.
- Одним из достоинств OpenMP его разработчики считают поддержку так называемых **"orphan" (оторванных) директив**, то есть директивы синхронизации и распределения работы могут не входить непосредственно в лексический контекст параллельной области.

Еще раз 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/).