Белгородский государственный технологический университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №2 по теме: «Решение практических задач с применением технологии OpenMP»

**Выполнил:**  
студент группы ПВ-31  
Адаменко И. И.

**Проверил:**к. т. н., доцент  
Михелёв В. М.

Белгород  
2015

**Цель работы:** изучить опции директив OpenMP; получить практический навык использования технологии OpenMP при решении прикладных задач.

# Теоретическая часть

OpenMP — это набор директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с единой памятью на языках C, C++ и Fortran. OpenMP поддерживается многими коммерческими компиляторами на различных платформах. В настоящее время опубликована спецификация OpenMP версии 4.0.

OpenMP прост в использовании и включает лишь два базовых типа конструкций: директивы pragma и функции исполняющей среды OpenMP, которые подключаются как дополнительная библиотека. Директивы pragma, как правило, указывают компилятору реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с #pragma omp. Как и любые другие директивы pragma, они игнорируются компилятором, не поддерживающим конкретную технологию — в данном случае OpenMP.

Функции OpenMP служат в основном для изменения и получения параметров среды. Кроме того, OpenMP включает API–функции для поддержки некоторых типов синхронизации. Чтобы задействовать эти функции библиотеки OpenMP времени выполнения (исполняющей среды), в программу нужно включить заголовочный файл omp.h. Если в приложении используются только OpenMP–директивы pragma, включать этот файл не требуется.

Значительная часть функциональности OpenMP реализуется при помощи директив компилятору. Они должны быть явно вставлены пользователем, что позволит выполнять программу в параллельном режиме. Директивы OpenMP в программах оформляются указаниями препроцессору, начинающимися с #pragma omp. Ключевое слово omp используется для того, чтобы исключить случайные совпадения имён директив OpenMP с другими именами.

Формат директивы на Си/Си++:

#pragma omp directive-name [опция[[,] опция]...]

Объектом действия большинства директив является один оператор или блок, перед которым расположена директива в исходном тексте программы. В OpenMP такие операторы или блоки называются ассоциированными с директивой. Ассоциированный блок должен иметь одну точку входа в начале и одну точку выхода в конце. Порядок опций в описании директивы несущественен, в одной директиве большинство опций может встречаться несколько раз. После некоторых опций может следовать список переменных, разделяемых запятыми.

Все директивы OpenMP можно разделить на 3 категории: определение параллельной области, распределение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов – опций (clause). Отдельно специфицируются опции для назначения классов переменных, которые могут быть атрибутами различных директив. Чтобы задействовать функции библиотеки OpenMP периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл omp.h.

Все функции, используемые в OpenMP, начинаются с префикса omp\_. Если пользователь не будет использовать в программе имён, начинающихся с такого префикса, то конфликтов с объектами OpenMP заведомо не будет. В языке Си, кроме того, является существенным регистр символов в названиях функций. Названия функций OpenMP записываются строчными буквами. Для того чтобы программа, использующая функции OpenMP, могла оставаться корректной для обычного компилятора, можно прилинковать специальную библиотеку, которая определит для каждой функции соответствующую «заглушку» (stub).

# Практическая часть

## Задание для варианта №1

Численно найти значение определённого интеграла данной функции с точностью до 10 знаков после запятой.

## Описание алгоритма

В качестве функции была выбрана .

Пусть необходимо найти значение интеграла .

Основная идея большинства методов численного интегрирования состоит в замене подынтегральной функции на более простую, интеграл от которой легко вычисляется аналитически. При этом для оценки значения интеграла получаются формулы вида: ,

где — число точек, в которых вычисляется значение подынтегральной функции. Точки называются узлами метода, числа — весами узлов.

Одним из методов численного интегрирования является методов Симпсона, в котором подынтегральная функция на отрезке интегрирования заменяется параболой. Обычно в качестве узлов метода используют концы отрезка и его среднюю точку. В этом случае формула имеет очень простой вид: .

Если разбить интервал интегрирования на 2N равных частей, то получим:

где — значение функции в i-ой точке.

Для распараллеливания цикла используется директива #pragma omp parallel for, для задания количество параллельных потоков функция omp\_set\_num\_threads(), для расчёта времени работы программы функция omp\_get\_wtime().

Блок-схема последовательного алгоритма изображена на рис. 1, а параллельного на рис. 2.

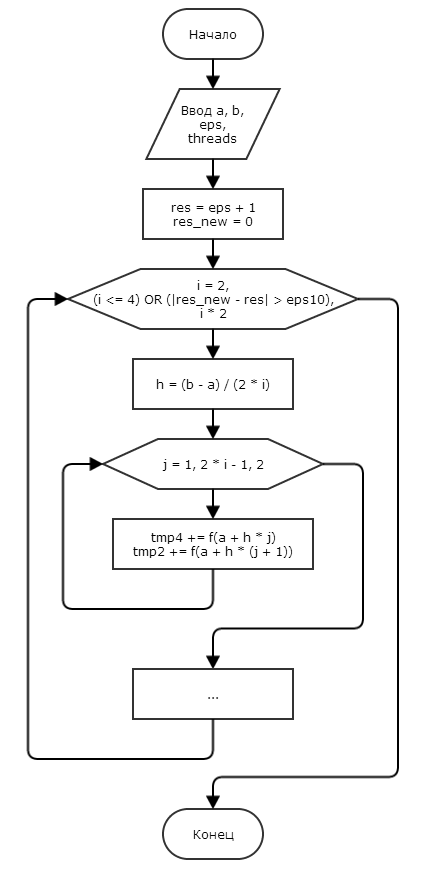


Рисунок 1. Блок-схема последовательного алгоритма

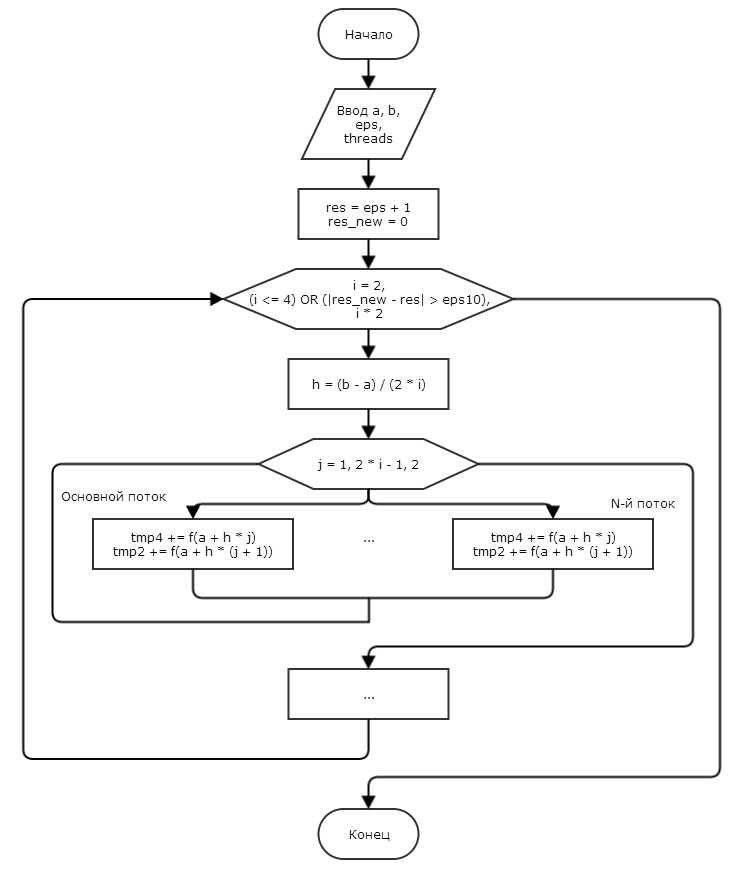


Рисунок 2. Блок-схема параллельного алгоритма

## Исходный код

1. #include <iostream>
2. #include <iomanip>
3. #include <stdlib.h>
4. #include <math.h>
5. #include <omp.h>
7. **using** **namespace** std;
9. **long** **double** f(**long** **double** x) {
10. **return** sqrt(fabs(cos(tan(sin(x \* x \* x + x \* x + 2 \* x))))) \* pow(x, 2);
11. }
13. **int** main(**int** argc, **char** **const** \*argv[]) {
14. **long** **double** a, b, eps, eps10, threads;
16. **if** (argc == 5) {
17. a = strtold(argv[1], NULL);
18. b = strtold(argv[2], NULL);
19. eps = strtold(argv[3], NULL);
20. threads = strtold(argv[4], NULL);
21. } **else** {
22. cout << "a: ";
23. cin >> a;
24. cout << "b: ";
25. cin >> b;
26. cout << "eps: ";
27. cin >> eps;
28. cout << "threads: ";
29. cin >> threads;
30. }
32. eps10 = pow(10, -eps);
34. **long** **double** res = eps10 + 1, res\_new = 0, fA = f(a), fB = f(b);
35. **long** **int** total = 0;
37. **double** timer = omp\_get\_wtime();
39. **for** (**int** i = 2; (i <= 4) || (fabs(res\_new - res) > eps10); i \*= 2) {
41. **long** **double** h = (b - a) / (2 \* i), sum2 = 0, sum4 = 0, sum = 0, tmp4 = 0, tmp2 = 0;
43. omp\_set\_num\_threads(threads);
45. #pragma omp parallel for reduction(+:tmp4, tmp2) shared(a, h, i)
46. **for** (**int** j = 1; j <= 2 \* i - 1; j += 2)
47. {
48. tmp4 += f(a + h \* j);
49. tmp2 += f(a + h \* (j + 1));
50. }
52. total += 2 \* i - 1;
54. sum4 += tmp4;
55. sum2 += tmp2;
57. sum = fA + 4 \* sum4 + 2 \* sum2 - fB;
58. res = res\_new;
59. res\_new = (h / 3) \* sum;
60. }
62. timer = omp\_get\_wtime() - timer;
64. // cout << std::setprecision(eps) << res\_new << endl;
65. cout << total << ' ' << timer << endl;
67. **return** 0;
68. }

## Пример работы программы

* N — количество итераций алгоритма;
* Ti — время работы программы на i потоках, в секундах.

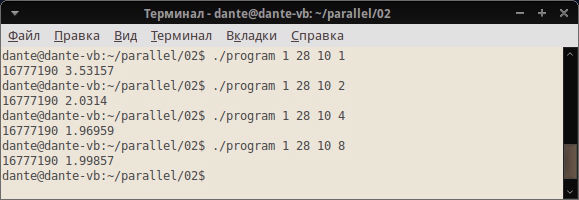
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **T­1** | **T2** | **T4** | **T8** |
| 52 500 | 0,014 | 0,013 | 0,011 | 0,012 |
| 2 100 000 | 0,463 | 0,281 | 0,274 | 0,272 |
| 8 400 000 | 2,025 | 1,254 | 1,191 | 1,164 |
| 67 800 000 | 15,473 | 8,601 | 8,825 | 8,348 |
| 271 200 000 | 59,760 | 33,304 | 32,513 | 32,366 |

Рассчитаем коэффициент ускорения для каждой группы итераций, по формуле:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **K­1** | **K2** | **K4** | **K8** |
| 52 500 | 1 | 1,04 | 1,22 | 1,12 |
| 2 100 000 | 1 | 1,64 | 1,68 | 1,70 |
| 8 400 000 | 1 | 1,61 | 1,69 | 1,73 |
| 67 800 000 | 1 | 1,79 | 1,75 | 1,65 |
| 271 200 000 | 1 | 1,79 | 1,83 | 1,84 |

## Скриншот программы

Пример работы программы при вычислении определённого интеграла на отрезке от 1 до 28, с точностью до 10 знаков после запятой, на 1, 2, 4 и 8 потоках.



## Вывод

На графике выше явно заметна зависимость, что чем больше итераций производится, тем выгоднее использовать параллельные вычисления. Во время выполнения работы пришлось пожертвовать эффективностью алгоритма вычисления в пользу эффективности распараллеливания, и, как видно, производительность программы от этого только увеличилась.

Также экспериментальным путём было выведено, что с повышением алгоритмической сложности каждой итерации, повышается и эффективность использования параллельных вычислений, поскольку если каждая итерация состоит лишь из элементарных математических вычислений, то на распараллеливание потоков уходит больше времени, чем на обработку итераций. Если же итерации алгоритмически сложны, то время, затрачиваемое на распараллеливание, практически не влияет на общее время работы программы.