Белгородский государственный технологический университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №1 по теме: «Знакомство с технологией OpenMP»

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

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

Белгород  
2015

**Цель работы:** ознакомиться с технологией OpenMP, научиться компилировать OpenMP-программы, изучить директивы технологии.

# 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.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создаёт набор «подчинённых» (slave) потоков, и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами, причём количество процессоров не обязательно должно быть больше или равно количеству потоков.

Важным достоинством технологии OpenMP является возможность реализации так называемого *инкрементального программирования,* когда программист постепенно находит участки в программе, содержащие ресурс параллелизма, с помощью предоставляемых механизмов делает их параллельными, а затем переходит к анализу следующих участков. Таким образом, в программе не распараллеленная часть постепенно становится всё меньше. Такой подход значительно облегчает процесс адаптации последовательных программ к параллельным компьютерам, а также отладку и оптимизацию.

# Практическая часть

## Задание для варианта №1

Протабулировать функцию на заданном отрезке с заданным шагом.

## Описание алгоритма

В качестве функции была выбрана .

Последовательный алгоритм решения задачи состоит в циклическом увеличении параметра на заданный пользователем шаг на отрезке с последующим вычислением функции. Однако, поскольку в задаче используется накапливающийся параметр (), для её распараллеливания было бы удобнее на каждом шаге вычислять его заново, тем самым значение параметра перестаёт напрямую зависеть от последовательности инкрементации , а значит распараллеливание становится более эффективным.

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.

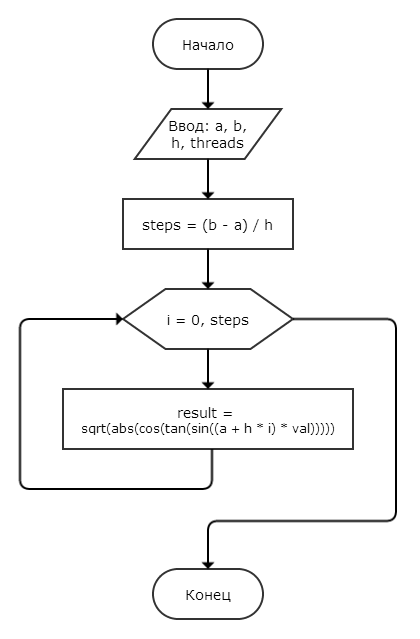


Рисунок 1. Блок-схема последовательного алгоритма

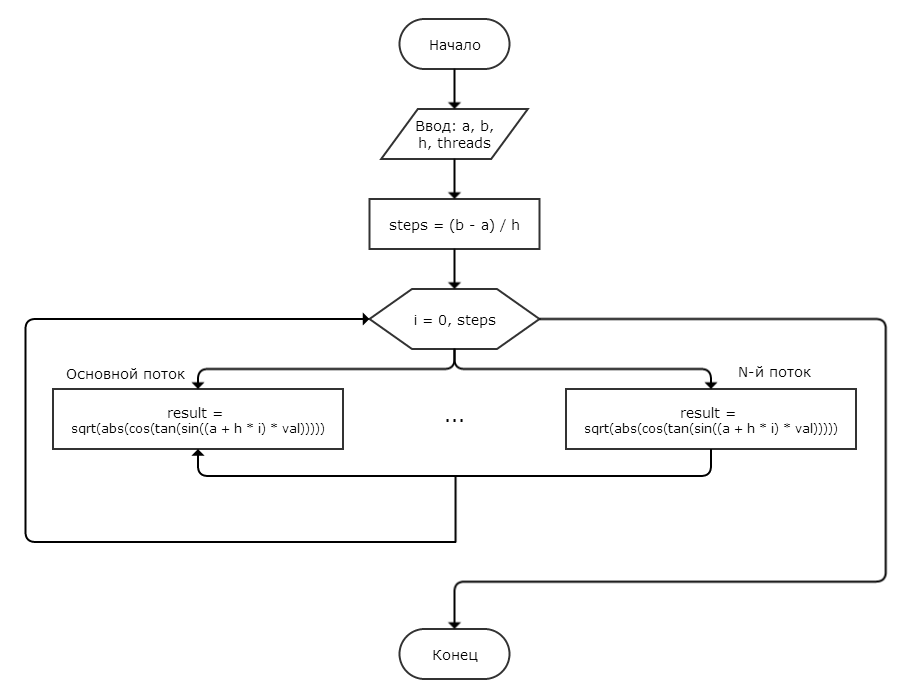


Рисунок 2. Блок-схема параллельного алгоритма

## Исходный код

|  |  |
| --- | --- |
| 01 | **#include** <stdio.h> |
| 02 | **#include** <stdlib.h> |
| 03 | **#include** <math.h> |
| 04 | **#include** <omp.h> |
| 05 |  |
| 06 | **#ifndef** M\_PI |
| 07 |  |
| 08 | **#define** M\_PI 3.14159265358979323846264338327 |
| 09 |  |
| 10 | **#endif** |
| 11 |  |
| 12 | **int** main(**int** argc, **char** **const** \*argv[]) |
| 13 | { |
| 14 | **long** **double** a, b, h, threads, val = M\_PI / 180; |
| 15 |  |
| 16 | **if** (argc == 5) { |
| 17 | a = strtold(argv[1], NULL); |
| 18 | b = strtold(argv[2], NULL); |
| 19 | h = strtold(argv[3], NULL); |
| 20 | threads = strtold(argv[4], NULL); |
| 21 | } **else** { |
| 22 | printf(*"a: "*); |
| 23 | scanf(*"%lf"*, &a); |
| 24 | printf(*"b: "*); |
| 25 | scanf(*"%lf"*, &b); |
| 26 | printf(*"h: "*); |
| 27 | scanf(*"%lf"*, &h); |
| 28 | printf(*"threads: "*); |
| 29 | scanf(*"%d"*, &threads); |
| 30 | } |
| 31 |  |
| 32 | **long int** steps = (b - a) / h; |
| 33 | **long** **double** result; |
| 34 |  |
| 35 | omp\_set\_num\_threads(threads); |
| 36 |  |
| 37 | **double** timer = omp\_get\_wtime(); |
| 38 |  |
| 39 | **#pragma omp parallel for shared(h, val, a, steps)** |
| 40 | **for** (**long** **int** i = 0; i < steps; i++) { |
| 41 | result = sqrt(abs(cos(tan(sin((a + h \* i) \* val))))); |
| 42 | } |
| 43 |  |
| 44 | timer = omp\_get\_wtime() - timer; |
| 45 |  |
| 46 | printf(*"Time: %lf\n"*, timer); |
| 47 |  |
| 48 | **return** 0; |
| 49 | } |

## Пример работы программы

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **T­1** | **T2** | **T4** | **T8** |
| 80 000 | 0,012 | 0,011 | 0,008 | 0,008 |
| 800 000 | 0,114 | 0,070 | 0,049 | 0,052 |
| 8 000 000 | 1,143 | 0,707 | 0,483 | 0,484 |
| 80 000 000 | 11,717 | 7,863 | 5,946 | 6,110 |
| 800 000 000 | 122,405 | 92,248 | 72,322 | 75,110 |

Рассчитаем коэффициент ускорения для каждой группы итераций, по формуле:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **K­1** | **K2** | **K4** | **K8** |
| 80 000 | 1 | 1,08 | 1,56 | 1,52 |
| 800 000 | 1 | 1,61 | 2,32 | 2,18 |
| 8 000 000 | 1 | 1,62 | 2,37 | 2,36 |
| 80 000 000 | 1 | 1,49 | 1,97 | 1,91 |
| 800 000 000 | 1 | 1,32 | 1,69 | 1,63 |

## 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.