КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ФИЗИКИ

Кафедра радиоастрономии

О.Г. Хуторова

ОСНОВЫ РАБОТЫ С БИБЛИОТЕКОЙ OPENMP

Учебно-методическое пособие

Казань-2022

УДК 004.032.24

Принято на заседании учебно-методической комиссии Института физики протокол № 1 от 29 октября.2022

Рецензент:

кандидат технических наук,

доцент кафедры компьютерных систем ФГБОУ ВО «КНИТУ-КАИ» Р.К. Классен

Хуторова О.Г.

Основы работы с библиотекой OpenMP

Учебно-методическое пособие / Хуторова О.Г. – Казань: Казан. ун-т., 2022. – 26 с.

Аннотация: Методическое пособие предназначено для обучающихся по программе «Введение в высокопроизводительные вычислительные системы» и «Многопоточная обработка больших данных». Пособие включает в себя описание основных функций и директив библиотеки OpenMP, особенностей их работы и практические задания для освоения программирования многопроцессорных систем. Может быть полезно студентам, аспирантам, слушателям ФПК.

Оглавление

Применение OpenMP

Важной составляющей деятельности в области физики, астрономии, защиты информации является применение различных численных методов для обработки больших массивов данных. Эта задача требует грамотного применения многоядерных вычислений. Библиотека OpenMP часто используется для распараллеливания математических вычислений из-за простоты использования и благодаря тому, что OpenMP — это открытый стандарт. OpenMP разработана для параллельного программирования вычислительных систем с общей памятью, например, многоядерных компьютеров. Официально поддерживается в языках C, C++, Фортран. Есть примеры применения этого стандарта в языках Pascal и Java. Все примеры в данном пособии написаны на языке C.

Компиляторы, поддерживающие OpenMP:

- GCC, начиная с версии 4.2
- IBM XL C/C++ Compiler
- Intel Fortran and C/C++ Compilers, начиная с версии 11.0 и 11.1
- Intel Parallel Studio
- Nanos compiler
- Portland Group compilers
- Qt, начиная с версии 4.2
- Sun Studio
- Visual C++, начиная с версии 2005 (Professional and Team System editions)

Подключение библиотеки

Для использования библиотеки OpenMP необходимо подключить заголовочный файл с помощью директивы include

```
#include <omp.h>
```

При сборке проекта, необходимо добавить опцию сборки:

```
-fopenmp
```

В терминале Linux это выглядит так:

```
gcc name.c –o name.exe –fopenmp
```

Компиляция и сборка в MS Visual Studio требует установить соответствующие флажки в настройках проекта (для Visual Studio) для подключения библиотек стадии компиляции: vcomp.lib и vcompd.lib; библиотеки времени выполнения: vcomp90.dll и vcomp90d.dll и

Включить

```
OpenMP:Project → %name% Properties
    Configuration Properties → C/C++ → Language
        «OpenMP Support»
```

Компиляция и сборка в Qt кроме подключения заголовочного файла omp.h требует изменения файла проекта .pro, надо добавить в него строки

```
QMAKE_CXXFLAGS+= -fopenmp

QMAKE_LFLAGS += -fopenmp
```

Параллельные потоки

Библиотека OpenMP использует параллелизм потоков. Параллельная программа состоит из последовательных и параллельных секций. После запуска программы создается единственный процесс, который начинает выполняться как обычная последовательная программа – это главная нить (*master*), выпол-

няющая последовательные секции программы. При входе в параллельную секцию выполняется операция *fork*, порождающая семейство нитей. Каждая нить имеет свой уникальный числовой идентификатор (главной нити соответствует 0). При распараллеливании циклов все параллельные нити исполняют один код. В общем случае нити могут исполнять различные фрагменты кода. При выходе из параллельной секции выполняется операция *join*. Завершается выполнение всех нитей, кроме главной (Рис.1).

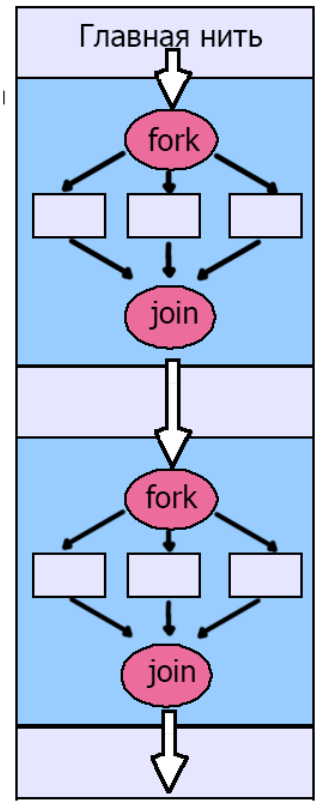

Рисунок 1 Последовательные и параллельные области программы с библиотекой OpenMP

Директива parallel

Создание параллельной области осуществляется директивой компилятора parallel

```
#pragma omp parallel
```

Границы параллельной области выделяются фигурными скобками. В начале параллельной области главным потоком, который существует все время работы процесса, порождается ряд потоков, их число можно задать явно, по умолчанию будет создано столько потоков, сколько в системе вычислительных ядер. В конце параллельного региона потоки неявно синхронизируются и уничтожаются.

Функции библиотеки

- omp_get_thread_num() – возвращает номер текущего потока,
- omp_ get _num_threads() – возвращает число потоков.

Пример:

```
#include <stdio.h>
#include <omp.h>
int main()
{
printf("Single thread 1\n");
#pragma omp parallel
 {
 printf("Parallel thread, %d\n", omp_get_thread_num());
 }
printf("Single thread 2\n");
}
```

В главной нити на консоль выводится «Single thread 1», далее происходит распараллеливание и каждый поток печатает «Parallel thread» и свой номер, затем опять главная нить на консоль выводит «Single thread 2». Так как в программе явно не задано число потоков, то оно будет равно числу ядер компьютера.

Локальная и разделяемая память

Потоки создаются на основе одного процесса, имеют как собственную локальную память, так и доступ к общей памяти.

Все переменные, созданные до директивы parallel, являются общими для всех потоков. Переменные, созданные внутри потока, являются локальными (приватными) и доступны только текущему потоку (Рис.2).

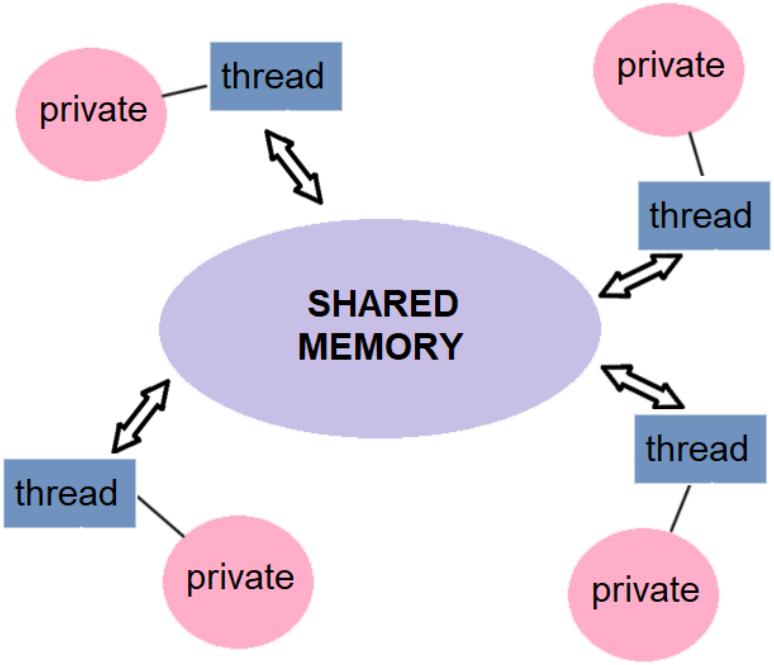

Рисунок 2 Общая и локальная память потоков программы

Для того, чтобы задать список переменных, для которых порождается локальная копия в каждой нити, используется опция директивы parallel

```
private(список переменных)
```

Пример:

```
int X=1;
#pragma omp parallel private(X)
{
  X += omp_get_thread_num();
}
```

В примере при создании параллельных нитей каждая нить получает свою копию переменной X, а затем прибавляет к ней свой уникальный номер с помощью функции omp_get_thread_num().

Для того, чтобы использовать локальные переменные каждой нити, после параллельной секции применяется опция

```
reduction (оператор: список)
```

Оператор задает выполняемое действие над приватными переменными, возвращаемыми из каждой нити в главную нить. Для каждой переменной будет создана локальная копия переменной, которая инициализируется соответственно типу оператора (для аддитивных операций – 0, для мультипликативных – 1). Над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор. Операторы C/C++: -, +, *, &, |, ,^ &&, ||.

Пример:

```
#include <stdio.h>
#include <omp.h>
int main() {
int count = 0;
#pragma omp parallel reduction (+: count)
 {
      count++;
      printf("count = %d ", count);
 }
printf( " thread count = %d ", count);
return 0;
}
```

В программе каждая нить получает свою копию переменной count, увеличивает ее на 1, по выходу из параллельной области переменные складываются, главная нить получает в общую переменную count сумму локальных переменных, равную числу процессоров. В случае оператора * переменные перемножаются,

Другие опции директивы parallel

- num_threads (int) – явно задаёт количество нитей;
- if(условие) – выполнение параллельной области по условию;
- shared(список) – явно задаёт список переменных, общих для всех нитей;

! Количество потоков нельзя изменять внутри параллельной секции

Директивы parallel могут быть вложенными, при этом могут создаваться вложенные потоки. Функция omp_set_nested (int) разрешает или запрещает вложенный параллелизм.

Пример:

```
omp_set_nested(1); //разрешение вложенного параллелизма
#pragma omp parallel num_threads(3) private(n) // 3 нити
 {
 n = omp_get_thread_num();
 #pragma omp parallel // вложенная область
  {
   printf("Thread %d - %d\n", n, omp_get_thread_num());
  }
 }
```

В примере вложенная область имеет число нитей, равное числу ядер процессора. Общее число строк в консоли будет равно утроенному числу ядер.

Директива for

Используется для явного распараллеливания следующего цикла for внутри параллельной секции, при этом каждая нить начнет со своего индекса. Если не указывать директиву, то цикл будет пройден каждой нитью полностью от начала и до конца. Данная директива просто размещается перед циклом.

```
#pragma omp for [опция1, опция2]
for(int i = 0; i < 100; i++)
    {…}
```

Директива parallel for

Если в программе требуется распараллелить только цикл, то используют директиву

```
#pragma omp parallel for [опция [[,] опция]…]
```

Опции этой директивы такие же, как у директивы parallel (private, shared, reduction).

Для того, чтобы оптимизировать параллельный цикл, важно адекватно использовать следующие опции.

- Опция schedule(type[, chunk]) тип распределения итераций в цикле по потокам
    - static – блоки равного размера chunk, если размер chunk не указан, то множество итераций цикла делится на приблизительно равного размера непрерывные куски. Данная опция стоит по умолчанию

- dynamic – освободившиеся потоки забирают порцию итераций размером chunk (по умолчанию = 1)
- guided – подобно dynamic, но размер порции итераций постепенно уменьшается с некоторого начального значения до chunk пропорционально количеству ещё не распределенных итераций
- auto – компилятор сам выбирает способ
- runtime – определяется переменной среды

Директива ordered определяет блок внутри тела цикла, который должен выполнятся в том порядке, в котором итерации идут в последовательном цикле

```
#pragma omp ordered
    {
    <code>
    }
```

Опция nowait – Если клауза nowait не указана, то конструкция for неявно завершится барьерной синхронизацией. В конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих потоков: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция nowait позволяет потокам, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными.

Правила безопасности OpenMP для циклов

- Параметр цикла должен иметь тип integer.
- Операция сравнения должна иметь формат:
  Параметр цикла <, <=, >, >= инвариант_целого_типа
- Если используется операция сравнения < или <=, параметр цикла должен увеличиваться при каждой итерации, а при использовании операции > или >= параметр цикла должен уменьшаться.
- Третье выражение (или инкрементная часть цикла for) должно являться либо целочисленным сложением, либо целочисленным вычитанием и должно практически совпадать со значением инварианта цикла.
- Цикл должен являться базовым блоком. Это означает, что не разрешены переходы из цикла, за исключением оператора exit, который завершает работу всего приложения. Если используются операторы goto или break, они должны приводить к переходам внутри цикла, а не вне его.

Синхронизация потоков

При изменении общей переменной одновременно несколькими потоками, или когда один поток пытается читать переменную в то время, как другой ее изменяет, возникает так называемое состояние гонки за данными. В этом случае невозможно гарантировать порядок записи и, соответственно, результат. Чтобы не допускать гонки за данными, потоки синхронизируют с помощью директив critical, atomic, barrier или с помощью специальных переменных - замков.

Директива critical

С помощью директивы critical оформляется критическая секция программы. Данная секция может выполнятся только одним потоком за раз. Если критическая секция уже выполняется каким-либо потоком, то другие потоки будут

заблокированы, пока выполняющий секцию поток не закончит её выполнение. Затем один из заблокированных потоков начнет выполнять критическую секцию, а остальные заблокированные продолжать ожидание. Секции желательно именовать! Иначе они условно будут ассоциироваться с одним и тем же именем.

Пример:

```
#include <omp.h>
#include <stdio.h>
int main() {
int value = 0;
#pragma omp parallel for num_threads(8)
    for(int i = 0; i < 100; i++)
    {
    #pragma omp critical (name1) // именование сек-
ции
        {
        value = value + 1;
        }
    }
printf("value = %d\n", value);
return 0;
}
```

В примере выводимое значение должно быть равено 100, без директивы critical значение переменной `value` непредсказуемо.

Для ряда операций более эффективно использовать директиву atomic, чем критическую секцию. Она ведет себя так же, но работает чуть быстрее.

Директива barrier

Нити, выполняющие текущую параллельную область, дойдя до директивы

```
barrier,
```

останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше.

Пример:

```
#include <stdio.h>
#include <omp.h>
int main()
{

#pragma omp parallel
    {
    printf("String 1\n");
    printf("String 2\n");

    #pragma omp barrier
    printf("String 3\n");
    }
}
```

В примере все потоки дойдут до печати «`String 3`» только после того, как напечатают «`String 1`» и «`String 2`», печать первых двух строк будет происходить в непредсказуемом порядке.

Замки

Замки (locks) обеспечивают явный механизм синхронизации потоков. Замок может находится в трех состояниях: неинициализирован, разблокирован, заблокирован. Разблокированный замок может быть захвачен потоком, при этом он блокируется. Только поток захвативший замок может его разблокировать.

omp_init_lock() и omp_init_nest_lock() – инициализируют простой и множественный замок соответственно

omp_destroy_lock() и omp_destroy_nest_lock() – уничтожают замок

omp_set_lock() и omp_set_nest_lock() – блокируют

omp_unset_lock() и omp_unset_nest_lock() –разблокируют

omp_test_lock() и omp_test_nest_lock() – проверяет, заблокирован ли замок, при этом не блокируя поток

Пример:

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
omp_lock_t lock; // объявляем переменную замка
int main()
{
int n;
omp_init_lock(&lock);       // инициализируем замок
#pragma omp parallel private(n)
    {
    n = omp_get_thread_num();
    omp_set_lock(&lock);    // блокировка
    printf("Begin %d\n", n);
```

```
    sleep(1);
    printf("End %d\n", n);
    omp_unset_lock(&lock); // разблокировка
    }
omp_destroy_lock(&lock);    // уничтожение замка
return 0;
}
```

В программе нет никаких проблем с синхронизацией. Потоки получают доступ к консоли в непредсказуемом порядке, но замок поставлен для того, чтобы каждый поток, получивший доступ, вывел сначала «Begin», а после паузы «End», не передавая консоль другому потоку.

! Нельзя использовать блокировку без инициализации замка

Параллельные секции

Если в программе появляется несколько фрагментов, не зависящих друг от друга, но имеющие зависимости внутри себя — то их распараллеливают с помощью механизма параллельных секций:

Пример:

```
#pragma omp parallel private(n)
{
n = omp_get_thread_num();

#pragma omp sections // begin sections

{
```

```
#pragma omp section
    {
    printf("Section 1 run Task1: %d\n", n);
    Task1 ();
    }
#pragma omp section
    {
    printf("Section 2 run Task2: %d\n", n);
    Task2 ();
    }

#pragma omp section
    {
    printf("Section 3 run Task3: %d\n", n);
    Task3 ();
    }
}     // end sections
printf("Parallel region: %d\n", n);
}
```

В параллельной области выделены три секции, в каждой секции выполняется своя функция: `Task1, Task2, Task3`. Эти функции выполняются параллельно разными потоками.

! Не пытайтесь распараллелить рекурсивные функции с помощью секций.

Директивы master и single

Директива master выделяет участок кода, который будет выполнен в параллельной области только нитью-мастером.

Определяет код, который выполняется только одной (первой пришедшей в данную точку) нитью. Остальные нити пропускают соответствующий код и ожидают окончания его выполнения. Если ожидание других нитей необязательно, может быть добавлен параметр nowait.

Пример:

```
void main()
{
omp_set_num_threads(3);
 #pragma omp parallel
 {
  printf("Message 1");
  #pragma omp single nowait
  {
  printf("Only one thread");
  }
  printf("Message 2");
 }
}
```

В примере сообщение `"Only one thread"` распечатает любая нить, которая первой выполнит более ранние операции.

Оценка времени выполнения кода

Самое главное, для чего нужна библиотека OpenMp, это ускорение выполнения вычислений; чтобы его оценить, применяются специальные функции - таймеры

omp_get_wtime(void) – возвращает время в секундах, прошедшее с произвольного момента в прошлом. Точка отсчета остается неизменной в течение всего времени выполнения программы.

omp_get_wtick() – точность таймера (зависит от платформы)

Пример:

```
// объявляем переменные для оценки времени
double start_time, end_time, tick;


// засекаем начало расчетов
start_time = omp_get_wtime();
// интересующий нас участок кода
      {……}
// засекаем конец расчетов
end_time = omp_get_wtime();
// узнаем точность таймера
tick = omp_get_wtick();
// выводим результат
printf("total time %f\n", end_time-start_time);
printf("timer's precision %f\n", tick);
```

Алгоритм выполнения лабораторных работ

Провести эксперимент по анализу ускорения вычислений на многоядерном компьютере. Работу провести в следующей последовательности.

a) Составить и отладить программу, решающую задачу, на языке C в обычном, последовательном режиме вычислений. Доказать, что программа выполняется верно.

b) Продумать декомпозицию задачи (функциональную или по данным).

c) Продумать алгоритм распараллеливания и коммуникации.

d) Найти функции библиотеки параллельного программирования, наиболее подходящие к выбранному алгоритму распараллеливания задачи.

e) Отладить программу, работающую в параллельном режиме.

f) Добавить в программы функции оценки времени выполнения из библиотеки OpenMP.

g) Увеличить объем вычислений в задачах (количество сигналов, размер массивов, временную сложность вычисления функций). Функцию можно задать как сумму других функций, например,

$$S = 1 + \frac{\cos(x)}{1!} + \frac{\cos(2x)}{2} + \ldots + \frac{\cos(nx)}{n!},$$

где n = 1000 (вместо cos может быть любая другая функция).

h) Провести несколько экспериментов по оценке времени выполнения задачи с различным количеством потоков.

i) Проанализировать и оценить результаты численных экспериментов.

j) Сделать выводы о влиянии на результат различных факторов (архитектуры вычислительной системы, эффективности программы, выбранных алгоритмов, сложности вычислений, времени коммуникаций).

k) Написать отчет. Отчет должен содержать цель работы, задачу работы, решение этой задачи на конкретной платформе (ОС, тип процессора, язык, компилятор и его настройка), выводы. Оформить отчет согласно ГОСТ 7.32-2017. Титульный лист обязателен. Листинги программ вставлять в виде текста.

Результаты работы вставлять как в виде скриншотов командной строки, так и в виде текста или таблицы. Скриншоты не обрезать. В выводах оценить ускорение, которое дает параллельная программа. Имя файла с отчетом должно быть фамилией.

Варианты задания:

1. С использованием библиотеки OpenMP составить программу вычисления суммы элементов вектора.
2. С использованием библиотеки OpenMP составить программу вычисления суммы двух векторов.
3. С использованием библиотеки OpenMP составить программу вычисления суммы элементов двумерного массива.
4. С использованием библиотеки OpenMP составить программу вычисления суммы двумерных массивов (вычислить матрицу C = A+B).
5. С использованием библиотеки OpenMP составить программу вычисления интеграла методом прямоугольников с заданной точностью.
6. С использованием библиотеки OpenMP составить программу вычисления интеграла методом трапеций с заданной точностью.
7. С использованием библиотеки OpenMP составить программу вычисления интеграла методом Симпсона с заданной точностью.
8. С использованием библиотеки OpenMP составить программу вычисления двумерного интеграла с заданной точностью.
9. С использованием библиотеки OpenMP составить программу вычисления матрицы C, как функцию от заданных матриц A и B, заполненных случайными числами: $C = A^2+B^2$.

10. С использованием библиотеки OpenMP составить программу вычисления матрицы C как функцию от заданных матриц A и B: $C = (A-B)^2$.
11. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = (B–A)+A^2$.
12. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = (A\cdot5 – B\cdot8) + A\cdot B$.
13. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = A^2- B^2$.
14. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = (A\cdot5 – B\cdot8) + A\cdot B$.
15. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = A\cdot B – 11$.
16. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = A\cdot B + A^2$.
17. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц $C = (B–A)^2–B^2$.
18. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = A\cdot B–B+3$.
19. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = B+3\cdot A–B^2$.

20. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = (A-B)^2 \cdot 18 – 3 \cdot A \cdot B$.
21. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = A^2+B \cdot 8 + 3$.
22. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = A \cdot B \cdot 8 + 3$.
23. С использованием библиотеки OpenMP вычислить матрицу C как функцию от заданных матриц A и B, заполненных случайными числами: $C = (A+B)^2$.
24. С использованием библиотеки OpenMP составить программу, реализующую дискретное преобразование Фурье сигнала (прямое и обратное), представление результата в виде амплитуд и фаз гармоник, сигнал – гармоника с заданной амплитудой и фазой.
25. С использованием библиотеки OpenMP составить программу, реализующую цифровой полосовой фильтр, сигнал – сумма гармоник и шума.
26. С использованием библиотеки OpenMP составить программу, реализующую вейвлет преобразование сигнала с материнской функцией Морле.
27. С использованием библиотеки OpenMP составить программу, реализующую вейвлет преобразование сигнала с материнской функцией Гаусса 1-го порядка.
28. С использованием библиотеки OpenMP составить программу, реализующую вейвлет преобразование сигнала с материнской функцией МНАТ.

29. С использованием библиотеки OpenMP составить программу, реализующую вейвлет преобразование сигнала с материнской функцией Хаара.
30. С использованием библиотеки OpenMP составить программу, реализующую вейвлет преобразование сигнала с материнской функцией Гаусса 4-го порядка.
31. С использованием библиотеки OpenMP составить программу, реализующую вейвлет преобразование сигнала с материнской функцией Добеши.
32. С использованием библиотеки OpenMP составить программу, реализующую вейвлет преобразование сигнала с материнской функцией Пауля.

ЛИТЕРАТУРА

1.The OpenMP API specification for parallel programming. – URL: https://www.openmp.org/ (дата обращения: 08.07.2022). – Режим доступа: свободный. – Текст: электронный.

2.Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие.– М.: Издательство Московского университета, 2012.-344 с.

3.Гергель В. П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем.- М.: Издательство Московского университета. – 2010. – С. 534.

4.Арыков, С. Б. Параллельное программирование над общей памятью. OpenMP : учебное пособие / С. Б. Арыков, М. А. Городничев, Г. А. Щукин. - Новосибирск : Изд-во НГТУ, 2019. - 95 с. - ISBN 978-5-7782-3796-4. - Текст : электронный. – URL: https://znanium.com/catalog/product/1866910 (дата обращения: 08.07.2022). – Режим доступа: по подписке.

5.Что такое OpenMP? URL: https://parallel.ru/tech/tech_dev/openmp.html (дата обращения: 06.05.2022). – Режим доступа: свободный. – Текст: электронный.

6.Васильев В.С. Учебник по OpenMP URL: https://pro-prof.com/archives/4335#page_2 (дата обращения: 06.05.2022). – Режим доступа: свободный. – Текст: электронный.