# Параллельные алгоритмы

## Лабораторная работа № 1

# Основные директивы OpenMP

### 1. Введение

Что такое OpenMP?

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP,ccNUMA, etc.) в модели общей памяти (shared memory model). В
стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды. Примерами систем с общей памятью, масштабируемых до большого числа процессоров, могут служить суперкомпьютеры Cray Origin2000 (до 128 процессоров), HP 9000 V-class (до 32 процессоров в одном узле, а в конфигурации из 4 узлов - до 128 процессоров), Sun Starfire (до 64 процессоров).

Где найти информацию?

Основной источник информации - сервер http://www.openmp.org/. На сервере оступны спецификации, статьи, учебные материалы, ссылки.

Базовый потоковый параллелизм

Общедоступный процесс памяти может состоять из множественных потоков, несколько нитей управления, которые имеют общее адресное пространство, но разные потоки команд и раздельные стэки. В простейшем случае, процесс состоит из одной нити. Нити иногда называют также потоками, легковесными процессами, LWP (light-weight processes).. OpenMP основан на существовании множественных потоков в общедоступной памяти, программирующей парадигму.

Явный Параллелизм

OpenMP имеет явную (не автоматический) модель программирования, предлагая программисту полное управление по распараллеливанию.

Модель Fork-Join (Ветвление – Объединение)

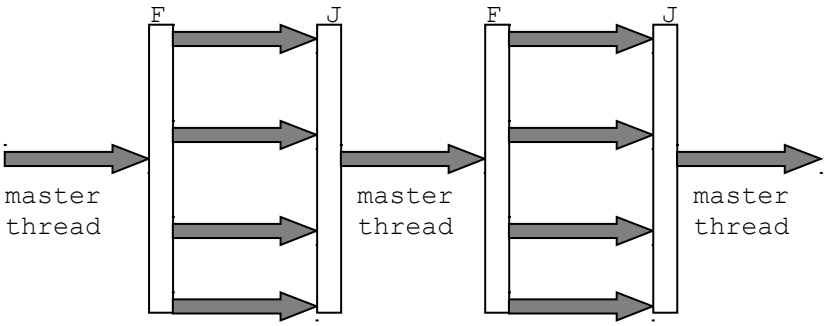
OpenMP использует Fork-Join модель параллельного выполнения:

Все программы OpenMP начинаются как единственный процесс: главный поток. Главный поток выполняется последовательно, пока не сталкиваются с первой областью параллельной конструкции.

Fork (ВЕТВЛЕНИЕ): главный поток создает группу параллельных потоков.

Инструкции в программе, которые включены параллельной конструкцией области{*региона*}, тогда выполнены параллельно среди различных потоков группы

Join (ОБЪЕДИНЕНИЕ): Когда потоки группы завершают инструкции в области параллельной конструкции, они синхронизируются и закрываются, оставляя только главный поток.




## 2. Директивы

Директивы OpenMP с точки зрения языка C являются командами препроцессора:

#pragma omp

Фортрана являются комментариями и начинаются с комбинации символов:

!$OMP.

Формат директивы на C/C++:

<команда для препроцессора> <имя директивы> <предложение(klausa)>

```
#pragma omp parallel [clause clause ...]
   {
    .
    .
    .
   }
```

Директивы можно разделить на 3 категории:

- определение параллельной секции,
- разделение работы,
- синхронизация.

Каждая директива может иметь несколько дополнительных атрибутов - клауз. Отдельно специфицируются клаузы для назначения классов переменных, которые могут быть атрибутами различных директив.

### Порождение нитей

```
#pragma omp parallel [clause clause ...]
  {
   .
   .
   .
  }
```

```
!$OMP PARALLEL ... END PARALLEL
```

clause: if(условие) – выполнение параллельной секции по условию;
private(list) – список переменных локальных в каждой нити;
shared(list) - список переменных общих для каждой нити;
firstprivate(list) - список переменных, которые становятся локальными
в каждой нити со значениями, ранее присвоиными
этим переменным;

copyin(list) - список переменных (массивов), которые определены #pragma omp threadprivate(list), и которые создаются в каждой нити;
reduction(operator:list) - список переменных, с которыми выполняются операции обобщенно по всем нитям.

list: - список переменных;
operator: - +,-,:,*,. . . .

Определяет параллельную область программы. При входе в эту область порождаются новые (N-1), образуется "команда" из N нитей, а порождающая нить получает номер 0 и становится основной нитью команды (т.н. "master thread"). При выходе из параллельной области основная нить дожидается завершения остальных нитей, и продолжает выполнение в одном экземпляре. Предполагается, что в SMP-системе нити
будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы).

Каким образом между порожденными нитями распределяется работа - определяется директивами DO,SECTIONS и SINGLE. Возможно также явное управление распределением
работы (а-ля MPI) с помощью функций, возвращающих номер текущей нити и общее число нитей. По умолчанию (вне этих директив), код внутри PARALLEL исполняется всеми нитями одинаково.
Параллельные области могут динамически вложенными. По умолчанию (если вложенный параллелизм не разрешен явно), внутренняя параллельная область исполняется одной нитью.

## **Разделение работы** (work-sharing constructs)

Параллельные циклы

```
#pragma omp parallel [clause clause ...]
  { . . .
    . . .
#pragma omp for [clause clause ...]
  { . . .
    . . .
  }
#pragma omp for [clause clause ...]
  { . . .
  }
 . . .
 . . .
 }
```

```
!$OMP DO ... [ENDDO]
```

```
clause:    schedule(type[,chink])
           ordered
           private(list)
           shared(list)
           firstprivate(list)
           lastprivate(list)
           reduction(operator:list)
           nowait
```

Определяет параллельный цикл.
Клауза schedule определяет способ распределения итераций по нитям:

  static,m - статически, блоками по m итераций
  dynamic,m - динамически, блоками по m (каждая нить берет на выполнение первый
  еще невзятый блок итераций)
  guided,m - размер блока итераций уменьшается экспоненциально до величины m
  runtime - выбирается во время выполнения.

ordered – (для циклов) реализуется последовательное выполнение витков цикла, как в последовательном алгоритме.
lastprivate(list) – переменным присваивается результат последнего витка цикла.
По умолчанию, в конце цикла происходит неявная синхронизация; эту синхронизацию
можно запретить с помощью nowait (ENDDO NOWAIT).

Параллельные секции

```
#pragma omp parallel [clause clause ...]
  { . . .
    . . .
#pragma omp sections [clause clause ...]
  { . . .
    . . .
#pragma omp section
  { . . .
    . . .
  }
#pragma omp section
   { . . .
     . . .
   }
  }     // - end sections
 . . .
 . . .
 }
```

```
!$OMP SECTIONS ... END SECTIONS
```

```
clause:    private(list)
           firstprivate(list)
           lastprivate(list)
           reduction(operator:list)
           nowait
```

Не-итеративная параллельная конструкция. Определяет набор независимых секций кода (т.н., "конечный" параллелизм). Секции отделяются друг от друга директивой
section.
Примечание. Если внутри parallel содержится только одна конструкция for или только одна конструкия section, то можно использовать укороченную запись: parallel for  или  parallel section.

Исполнение одной нитью

```
#pragma omp parallel [clause clause ...]
  { . . .
    . . .
#pragma omp single [clause clause ...]
  { . . .
    . . .
#pragma omp single [clause clause ...]
  { . . .
```

```
    . . .
  }
 }
```

```
!$OMP SECTIONS ... END SECTIONS

clause:    private(list)
           firstprivate(list)
           nowait
```

```
SINGLE ... END SINGLE
```

Определяет блок кода, который будет исполнен только одной нитью (первой, которая
дойдет до этого блока).

## Явное управление распределением работы

С помощью функций omp_get_thread_num() и omp_get_num_threads() нить может узнать свой номер и общее число нитей, а затем выполнять свою часть работы в зависимости от своего номера (этот подход широко используется в программах на базе интерфейса MPI).

**Пример**

Простой пример: вычисление числа "Пи". В последовательную программу вставлены две строчки, и она распараллелена!

C:

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

double f(double y)
  { return(4.0/(1.0+y*y));
  }

 main()
  { double w,x,sum,pi;
    int i;
    int n = 1000;
    w = 1.0/n;
    sum = 0.0;
#pragma omp parallel for schedule(static,n/2) private(i,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);

 return(0);
 }
```

Здесь переменная sum не должна описываться в private(i,x), но может быть описана в shared(w).

Fortran:

```
program compute_pi
```

```
   parameter (n = 1000)
   integer i
   double precision w,x,sum,pi,f,a
   f(a) = 4.d0/(1.d0+a*a)
   w = 1.0d0/n
   sum = 0.0d0;
!$OMP PARALLEL DO PRIVATE(x) SHARED(w) REDUCTION(+:sum)
   do i=1,n
       x = w*(i-0.5d0)
       sum = sum + f(x)
   enddo

   pi = w*sum
   print *,'pi = ',pi
   stop
   end
```

**Директивы синхронизации**

**#pragma omp master**

```
  {
   .
  }
```

MASTER ... END MASTER

Определяет блок кода, который будет выполнен только master-ом (нулевой нитью).
Вход/выход из критического интервала запрещены.

**#pragma omp critical[name]**

```
  {
   .
  }
```

CRITICAL ... END CRITICAL

Определяет критическую секцию, то есть блок кода, который не должен выполняться
одновременно двумя или более нитями.
name – имя критического интервала. Имя – глобальный идентификатор. Различные критические интервала с одним именем обрабатываются как одна и таже область (критический интервал).
Вход/выход из критического интервала запрещены.

**#pragma omp barrier**

BARRIER

Определяет точку барьерной синхронизации, в которой каждая нить дожидается всех
остальных. Наименьшая инструкция, которая содержит barrier должна быть структурным блоком. barrier не может быть в for и sections.

Неверно:

```
for(x ==0)
   #pragma omp barrier
```

Верно:

```
for(x ==0)
```

```
  {   #pragma omp barrier
  }
```

**#pragma omp atomic**
**  x binop = exper;**

```
 x++
++x
x--
--x
```

binop - +,*,-,/,&,^,\,>>,<<,or;
exper – скалярное выражение.

ATOMIC

Определяет переменную в левой части оператора "атомарного" присваивания, которая
должна корректно обновляться несколькими нитями.

**#pragma omp ordered**

ORDERED ... END ORDERED

Определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле. Может использоваться для упорядочения вывода от параллельных нитей.

**#pragma omp flush(list)**

FLUSH

Явно определяет точку, в которой реализация должна обеспечить одинаковый вид памяти для всех нитей. Неявно FLUSH присутствует в следующих директивах: BARRIER, CRITICAL, END CRITICAL, DO, END DO, PARALLEL, END PARALLEL, SECTIONS, END SECTIONS, SINGLE, END SINGLE, ORDERED, END ORDERED. В целях синхронизации можно также пользоваться механизмом замков (locks).

## 3. Классы переменных

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:
  SHARED (общие; под именем A все нити видят одну переменную) и
  PRIVATE (приватные; под именем A каждая нить видит свою переменную).
Отдельные правила определяют поведение переменных при входе и выходе из параллельной области или параллельного цикла: REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN.
По умолчанию, все COMMON-блоки, а также переменные, порожденные вне параллельной
области, при входе в эту область остаются общими (SHARED). Исключение составляют
переменные - счетчики итераций в цикле, по очевидным причинам. Переменные, порожденные внутри параллельной области, являются приватными (PRIVATE). Явно назначить класс переменных по умолчанию можно с помощью клаузы DEFAULT.

**shared(list)**

SHARED
Применяется к переменным, которые необходимо сделать общими.

**private(list)**

PRIVATE
Применяется к переменным, которые необходимо сделать приватными. При входе в параллельную область для каждой нити создается отдельный экземпляр переменной, который не имеет никакой связи с оригинальной переменной вне параллельной области.

#pragma omp **thredprivate(list)**

THREADPRIVATE
Применяется к COMMON-блокам, которые необходимо сделать приватными. Директива должна применяться после каждой декларации COMMON-блока.

```
double A[][],B[];
#pragma omp thredprivate(A,B)
```

**firstprivate**

FIRSTPRIVATE
Приватные копии переменной при входе в параллельную область инициализируются значением оригинальной переменной.

**lastprivate**

LASTPRIVATE
По окончании параллельно цикла или блока параллельных секций, нить, которая выполнила последнюю итерацию цикла или последнюю секцию блока, обновляет значение оригинальной переменной.

**reduction(+:A)**

REDUCTION(+:A)
Обозначает переменную, с которой в цикле производится reduction-операция (например, суммирование). При выходе из цикла, данная операция производится над копиями переменной во всех нитях, и результат присваивается оригинальной переменной. Переменные в этом списке не должны описываться в private(list), но могут быть описаны в shared(list).

**copyin(list)**

COPYIN
Применяется к COMMON-блокам, которые помечены как threadprivate. При входе в параллельную область приватные копии этих данных инициализируются оригинальными значениями.

## 4. Runtime-процедуры и переменные среды

В целях создания переносимой среды запуска параллельных программ, в OpenMP определен ряд переменных среды, контролирующих поведение приложения.
В OpenMP предусмотрен также набор библиотечных процедур, которые позволяют:

- во время исполнения контролировать и запрашивать различные параметры, определяющие поведение приложения (такие как число нитей и процессоров, возможность вложенного параллелизма);
- процедуры назначения параметров имеют приоритет над соотвествующими переменными среды.
- использовать синхронизацию на базе замков (locks).

**Переменные среды**

**export OMP_SCHEDULE "guided,4"**
**export OMP_SCHEDULE "dynamic"**

Определяет способ распределения итераций в цикле, если в директиве DO использована клауза SCHEDULE(RUNTIME).

**export OMP_NUM_THREADS=n** ( = 4, setenv )
Определяет число нитей для исполнения параллельных областей приложения.

**export OMP_DYNAMIC TRUE**
Разрешает или запрещает динамическое изменение числа нитей.

**export OMP_NESTED TRUE**
Разрешает или запрещает вложенный параллелизм.

## Процедуры для контроля/запроса параметров среды исполнения

**omp_set_num_threads(int num)**

OMP_SET_NUM_THREADS
Позволяет назначить максимальное число нитей для использования в следующей параллельной области (если это число разрешено менять динамически). Вызывается из последовательной области программы.

**omp_get_max_threads()**

OMP_GET_MAX_THREADS
Возвращает максимальное число нитей.

**omp_get_num_threads()**

OMP_GET_NUM_THREADS
Возвращает фактическое число нитей в параллельной области программы.

**omp_get_num_procs()**

OMP_GET_NUM_PROCS
Возвращает число процессоров, доступных приложению.

**omp_in_parallel()**

OMP_IN_PARALLEL
Возвращает .TRUE., если вызвана из параллельной области программы.

**omp_set_dunamic(int dynamic_threads)/omp_get_dinamic()**

OMP_SET_DYNAMIC / OMP_GET_DYNAMIC
Устанавливает/запрашивает состояние флага, разрешающего динамически изменять число нитей.

**omp_set_nested(int nested)/omp_get_nested()**

OMP_GET_NESTED / OMP_SET_NESTED
Устанавливает/запрашивает состояние флага, разрешающего вложенный параллелизм.

**Процедуры для синхронизации на базе замков**

В качестве замков используются общие переменные типа INTEGER (размер должен быть
достаточным для хранения адреса). Данные переменные должны использоваться только
как параметры примитивов синхронизации.

**omp_init_lock(omp_lock_t *lock)/ omp_destroy_lock(omp_lock_t *lock)**

OMP_INIT_LOCK(var) / OMP_DESTROY_LOCK(var)
Инициализирует замок, связанный с переменной var.

**omp_set_lock(omp_lock_t *lock)**

OMP_SET_LOCK(VAR)
Заставляет вызвавшую нить дождаться освобождения замка, а затем захватывает его.

**omp_unset_lock(omp_lock_t *lock)**

OMP_UNSET_LOCK(VAR)
Освобождает замок, если он был захвачен вызвавшей нитью.

**omp_test_lock(omp_lock_t *lock)**

OMP_TEST_LOCK(VAR)
Пробует захватить указанный замок. Если это невозможно, возвращает .FALSE.

## 5. Спецификация OpenMP для языков C/C++

Спецификация OpenMP для C/C++, выпущенная на год позже фортранной, содержит в основном аналогичную функциональность.
Необходимо лишь отметить следующие моменты:
1) Вместо спецкомментариев используются директивы компилятора "#pragma omp".
2) Компилятор с поддержкой OpenMP определяет макрос "_OPENMP", который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы.
3) Распараллеливание применяется к for-циклам, для этого используется директива
"#pragma omp for". В параллельных циклах запрещается использовать оператор break.
4) Статические (static) переменные, определенные в параллельной области программы, являются общими (shared).
5) Память, выделенная с помощью malloc(), является общей (однако указатель на нее может быть как общим, так и приватным).
6) Типы и функции OpenMP определены во включаемом файле <omp.h>.
7) Кроме обычных, возможны также "вложенные" (nested) замки - вместо логических
переменных используются целые числа, и нить, уже захватившая замок, при

повторном захвате может увеличить это число.

**Пример распараллеливания for-цикла в C**

```
#pragma omp parallel for private(i)
#pragma omp shared(x, y, n) reduction(+: a, b)
for (i=0; i<n; i++)
{
     a = a + x[i];
     b = b + y[i];
}
```

# Простые примеры параллельных программ в OpenMP

**Цель** – дать представление о построении простых параллельных программ на языке параллельного программирования OpenMP; практическое освоение основных директив языка.

## Методы распараллеливания и модели программ, поддерживаемые OpenMP

Базовый потоковый параллелизм

Общедоступный процесс памяти может состоять из множественных потоков, несколько нитей управления, которые имеют общее адресное пространство, но разные потоки команд и раздельные стэки. В простейшем случае, процесс состоит из одной нити. Нити иногда называют также потоками, легковесными процессами, LWP (light-weight processes).. OpenMP основан на существовании множественных потоков в общедоступной памяти, программирующей парадигму.

Явный Параллелизм

OpenMP имеет явную (не автоматический) модель программирования, предлагая программисту полное управление по распараллеливанию.

Модель Fork-Join (Ветвление – Объединение)

OpenMP использует Fork-Join модель параллельного выполнения:
Все программы OpenMP начинаются как единственный процесс: главный поток. Главный поток выполняется последовательно, пока не сталкиваются с первой областью параллельной конструкции.

Fork (ВЕТВЛЕНИЕ): главный поток создает группу параллельных потоков.

Инструкции в программе, которые включены параллельной конструкцией области{*региона*}, тогда выполнены параллельно среди различных потоков группы

Join (ОБЪЕДИНЕНИЕ): Когда потоки группы завершают инструкции в области параллельной конструкции, они синхронизируются и закрываются, оставляя только главный поток.

ПРИМЕР 1.1

Каждый процесс (ветвь п-программы) выводит на экран свой идентификационный номер и количество заказанных параллельных процессов.

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

main ()
{
int size, rank;

/* Создание множества параллельных процессов и в каждом из них задаются
 * свои приватные переменные size и rank */
#pragma omp parallel private(size, rank)
  {

/* Каждый процесс находит свой порядковый номер и выводит его на экран */
  rank = omp_get_thread_num();
  printf("Hello World from thread = %d\n", rank);

/* Главный процесс - master выводит на экран количество процессов */
  if (rank == 0)
    {
    size = omp_get_num_threads();
    printf("Number of threads = %d\n", size);
    }

  }  /* Завершение параллельной части */
}
```

## ПРИМЕР 1.2

Параллельное суммирование элементов двух векторов. Суммирование осуществляется в цикле. В программе применяется комбинация параллельного цикла и редуцированной операции по всем процессам.

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

main ()
{
int   i, n;
float a[100], b[100], sum;

/* Инициализация элементов векторов */
n = 100;
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
sum = 0.0;

/* Создание множества параллельных процессов и распараллеливание
 * цикла по виткам. При выходе из цикла все значения переменной sum
 * суммируются по всем процессам. */
 #pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);

/* Главный процесс выводит на экран значение sum */
printf("   Sum = %f\n",sum);
}
```

## ПРИМЕР 1.3

Пример аналогичный предыдущему: Параллельное суммирование элементов двух векторов. Но суммирование осуществляется в цикле в отдельной подпрограмме. В программе применяется комбинация параллельного цикла и редуцированной операции по всем процессам.

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

#define VECLEN 100
float a[VECLEN], b[VECLEN], sum;

/* Подпрограмма, в которой суммируются элементы векторов */
float dotprod ()
{
int i,rank;

rank = omp_get_thread_num();
#pragma omp for reduction(+:sum)
  for (i=0; i < VECLEN; i++)
    {
    sum = sum + (a[i]*b[i]);
    printf(" rank = %d i=%d\n",rank,i);
    }
return(sum);
}

main ()
{
int i;

/* Инициализация элементов векторов */
for (i=0; i < VECLEN; i++)
  a[i] = b[i] = 1.0 * i;
sum = 0.0;

/* Создание множества параллельных процессов */
#pragma omp parallel
  sum = dotprod();

printf("Sum = %f\n",sum);

}
```

## ПРИМЕР 1.4

Пример аналогичный примеру 3.2: Параллельное суммирование элементов двух векторов. Но суммирование осуществляется в отдельных секциях.

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

#define N     50

main ()
{
int i, size, rank;
float a[N], b[N], c[N];

/* Инициализация элементов векторов */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
```

```
/* Создание множества параллельных процессов */
#pragma omp parallel shared(a,b,c) private(i,rank,size)
  {

/* Каждый процесс находит свой порядковый номер и выводит его на экран */
  rank = omp_get_thread_num();
  printf("Thread %d starting...\n",rank);

/* Директива задания секций */
  #pragma omp sections nowait
    {
/* Секция 0*/
    #pragma omp section
      for (i=0; i < N/2; i++)
        {
        c[i] = a[i] + b[i];
        printf("rank = %d i= %d c[i]= %f\n", rank,i,c[i]);
        }
/* Секция 1*/
    #pragma omp section
      for (i=N/2; i < N; i++)
        {
        c[i] = a[i] + b[i];
        printf("rank = %d i= %d c[i]= %f\n", rank,i,c[i]);
        }

    }  /* Завершение блока секций */

  if (rank == 0)
    {
    size = omp_get_num_threads();
    printf("Number of threads = %d\n", size);
    }

  }  /* Завершение параллельной части */
}
```

## ПРИМЕР 1.5

Пример параллельного умножения матрицы на вектор.

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

#define M 10

main ()
{
float A[M][M], b[M], c[M];
int i, j, rank;

/* Инициализация данных */
for (i=0; i < M; i++)
  {
  for (j=0; j < M; j++)
    A[i][j] = (j+1) * 1.0;
  b[i] = 1.0 * (i+1);
  c[i] = 0.0;
  }
printf("\Вывод значений матрицы A и вектора b на экран:\n");
for (i=0; i < M; i++)
  {
```

```
  printf("  A[%d]= ",i);
  for (j=0; j < M; j++)
    printf("%.1f ",A[i][j]);
  printf("  b[%d]= %.1f\n",i,b[i]);
  }

/* Создание множества параллельных процессов и в каждом из них задаются
 * свои приватные переменные rank и i*/
#pragma omp parallel shared(A,b,c,total) private(rank,i)
  {
  rank = omp_get_thread_num();

/* Директива распараллеливания цикла по виткам */
#pragma omp for private(j)
  for (i=0; i < M; i++)
    {
    for (j=0; j < M; j++)
      c[i] += (A[i][j] * b[j]);

/* Каждый процесс выводит свой порядковый номер, значение витка цикла и
 * значение результирующего вектора на каждом витке цикла и внутри
 * критической секции */
    #pragma omp critical
      {
      printf(" rank= %d i= %d c[%d]=%.2f\n", rank,i,c[i]);
      }
    }   /* Конец параллельного цикла */
  } /* Завершение параллельной конструкции */
}
```

**ЗАДАНИЕ 1**

Тщательно изучить программы примеров. Скомпилировать и запустить все программы примеров.

# Задания к лабораторной работе № 1

Цель – дать представление о построении простых параллельных программ на языке параллельного программирования OpenMP; закрепить практическое освоение директив языка.

Задание 1. Проработка примеров из пункта "Примеры параллельных программ" этой же лабораторной.

Внимательно изучить примеры 1.1-1.5. Откомпилировать и запустить на 2-х процессах.

Задание 2. Параллельное умножение матрицы на вектор

**Вариант 1.**
Разработать алгоритм написать и отладить параллельную программу умножения матрицы на вектор с использованием распределения работ для параллельных процессов директивой **sections**. Использовать алгоритм примера 1.5.

**Вариант 2.**
Разработать алгоритм написать и отладить параллельную программу умножения матрицы на вектор с использованием распределения работ для параллельных процессов с непосредственным заданием работы («вручную»). Использовать алгоритм примера 1.5.

Задание 3. Параллельное умножение матрицы на матрицу.

**Вариант 1.**
Разработать алгоритм написать и отладить параллельную программу умножения матрицы на матрицу с использованием директивы распараллеливания цикла по виткам.

**Вариант 2.**
Разработать алгоритм написать и отладить параллельную программу умножения матрицы на матрицу с использованием распределения работ для параллельных процессов с непосредственным заданием работы («вручную»).

Задание 4. Сравнительные временные характеристики двух алгоритмов умножения матрицы на вектор на языке MPI и языке OpenMP.

В качестве параллельных программ взять параллельные программы: «умножение матрицы на вектор в MPI на топологии "кольцо"» и «умножение матрицы на вектор в OpenMP (один из вариантов)». Для обеих программ построить небольшие графики зависимости времени решения задачи от размеров матрицы и вектора. Обе программы запустить последовательно для матриц: A = [300 x 300], [500 x 500], [700 x 700], [1000 x 1000] и соответствующих этим матрицам векторов. Откомпилировать и запустить на 2-х компьютерах, засечь время и построить указанные графики.

# Контрольные вопросы к лабораторной работе №1

1. Методы распараллеливания и модели программ, поддерживаемые OpenMP.
2. Пояснить основные принципы выполнения ниже указанных директив.

Главная директива задания нитей:

```
#pragma omp parallel [clause clause ...]
 {
  . . .
 }
```

```
clause:   if(условие)
          private(list)
          shared(list)
          firstprivate(list)
          copyin(list)
          reduction(operator:list)
```

Директивы разделения работ по нитям

```
#pragma omp for [clause clause ...]
 {
   . . .
 }
```

```
clause:   schedule(type[,chink])
          ordered
          private(list)
          shared(list)
          firstprivate(list)
          lastprivate(list)
```

reduction(operator:list)
nowait

Параллельные секции

```
#pragma omp sections [clause clause ...]
 { . . .
   . . .
#pragma omp section
 { . . .
   . . .
 }
#pragma omp section
  { . . .
   . . .
  }
 }    // – end sections
```

clause: private(list)
firstprivate(list)
lastprivate(list)
reduction(operator:list)
nowait

Исполнение одной нитью

```
#pragma omp single [clause clause ...]
 {
   . . .
 }
```

clause: private(list)
firstprivate(list)
nowait

Явное управление распределением работы

Директивы синхронизации

```
#pragma omp master
 {
  . . .
 }
```

```
#pragma omp critical[name]
 {
  . . .
 }
```

```
#pragma omp barrier
```

```
#pragma omp atomic
```

3. Как задаются параллельные процессы при умножении матрицы на вектор и матрицы на матрицу в OpenMP?