Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Кемеровский государственный университет»
Новокузнецкий институт (филиал)

Факультет информатики, математики и экономики

Кафедра информатики и вычислительной техники им. В. К. Буторина

О.В. Михайлова, О.И. Новоселова

Технологии параллельного программирования

Методические указания к практическим занятиям для обучающихся по направлению подготовки

09.03.01 Информатика и вычислительная техника направленность (профиль) Автоматизированные системы обработки информации и управления)

Новокузнецк

2019

УДК [378.147.88:004.424](072)
ББК 74.484(2Рос-4Кем)я73+32.972я73
М69

Михайлова О.В.

М69 Технологии параллельного программирования : метод. указ. к практическим занятиям для обучающихся по направлению 09.03.01 - Информатика и вычислительная техника / О.В. Михайлова, О.И. Новоселова; Новокузнец. ин-т (фил.) Кемеров. гос. ун-та. – Новокузнецк : НФИ КемГУ, 2019. – 48 с.

Приведены основные теоретические сведения о технологии параллельного программирования OpenMP, рассмотрены примеры решения типовых задач с использованием данной технологии, приведены варианты индивидуальных заданий, даны рекомендации по их выполнению и оформлению отчетов о выполнении практических заданий в соответствии с требованиями НФИ КемГУ.

Предназначено для студентов, обучающихся по направлению 09.03.01 Информатика и вычислительная техника (профиль Автоматизированные системы обработки информации и управления). Может быть использовано студентами других направлений подготовки и специалистами, изучающими современные технологии программирования.

Рекомендовано
на заседании кафедры информатики и вычислительной техники
им. В. К. Буторина
13 февраля 2019 г.

Заведующий кафедрой
О. В. Михайлова

Утверждено
методической комиссией факультета информатики, математики и экономики
21 марта 2019 г.

Председатель методкомиссии
Г. Н. Бойченко

Содержание

Введение

Применение параллельных вычислительных систем (ПВС) является стратегическим направлением развития вычислительной техники. Это обстоятельство вызвано не только принципиальным ограничением максимально возможного быстродействия обычных последовательных ЭВМ, но и практически постоянным существованием вычислительных задач, для решения которых возможностей существующих средств вычислительной техники всегда оказывается недостаточно. Например, моделирование климата, генная инженерия, проектирование интегральных схем, анализ загрязнения окружающей среды, создание лекарственных препаратов и др., - требуют для своего анализа ЭВМ с производительностью более 1000 миллиардов операций с плавающей запятой в сек. (1 террафлопс).

Параллельные вычисления - это процессы обработки данных, в которых одновременно могут выполняться несколько операций компьютерной системы. [1]

Целями параллельных вычислений являются:

- сокращение времени исполнения программы;
- повышение конфигурируемости программы;
- возможно лучшая отказоустойчивость программного продукта;
- научный интерес.

Достижение параллелизма возможно только при выполнимости следующих требований:

- *независимость функционирования отдельных устройств* ***ЭВМ*** (устройства ввода-вывода, обрабатывающие процессоры, устройства памяти, и т.п.);
- *избыточность элементов вычислительной системы* в следующих основных формах: использование специализированных устройств (например, отдельные процессоры для целочисленной и вещественной арифметики, устройства многоуровневой памяти); дублирование устройств ЭВМ (например,

использование нескольких однотипных обрабатывающих процессоров или нескольких устройств оперативной памяти).

Возможные режимы выполнения независимых частей программы:

- *многозадачный режим* (режим разделения времени), при котором для выполнения нескольких процессов используется единственный процессор;
- *параллельное выполнение*, когда в один и тот же момент времени может выполняться несколько команд обработки данных;
- *распределенные вычисления*, при которых для параллельной обработки данных используется несколько обрабатывающих устройств, достаточно удаленных друг от друга, а передача данных по линиям связи приводит к существенным временным задержкам.

Одним из наиболее популярных средств программирования для компьютеров с общей памятью (см. Рисунок 1), базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология OpenMP. За основу берётся последовательная программа, а для создания её параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. Предполагается, что создаваемая параллельная программа будет переносимой между различными компьютерами с разделяемой памятью, поддерживающими OpenMP API. [2]

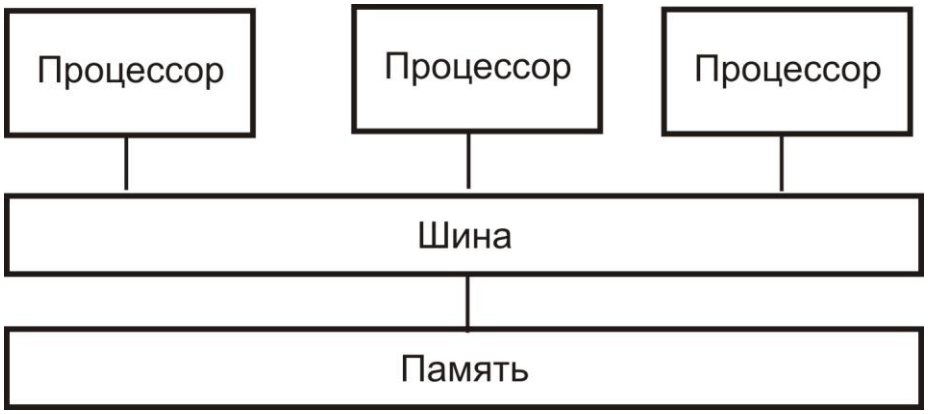


Рисунок 1 – Структура ЭВМ с разделяемой памятью

1 Базовые понятия и основные директивы OpenMP

OpenMP (Open specifications for Multi–Processing) – стандарт для написания параллельных программ для многопроцессорных вычислительных систем с общей оперативной памятью. Программа представляется как набор нитей (threads), объединённых общей памятью, где проблема синхронизации решается введением критических секций и мониторов.

Стандарт OpenMP был разработан в 1997 году, как API, ориентированный на написание портируемых многопоточных приложений. Сначала он был основан на языке Fortran, но позднее включил в себя и C/C++.

Разработкой стандарта занимается организация OpenMP ARB (ARchitecture Board), в которую вошли представители крупнейших компаний – разработчиков SMP-архитектур и программного обеспечения. Спецификации для языков Fortran и C/C++ появились соответственно в октябре 1997 года и октябре 1998 года. OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP, ccNUMA, etc.) в модели общей памяти (shared memory model). На данный момент последняя официальная спецификация стандарта – OpenMP 3.1 (принятая в июле 2011 года).

OpenMP – это набор специальных директив компилятору, библиотечных функций и переменных окружения. Наиболее оригинальны директивы компилятору, которые используются для обозначения областей в коде с возможностью параллельного выполнения. Компилятор, поддерживающий OpenMP, преобразует исходный код и вставляет соответствующие вызовы функций для параллельного выполнения этих областей кода.

За счет идеи “частичного распараллеливания” OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто добавляет в текст последовательной программы OpenMP директивы.

Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости одновременно 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.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» поток создает набор подчиненных потоков и задача распределяется между ними (см. Рисунок 2). Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно числу потоков). [3]

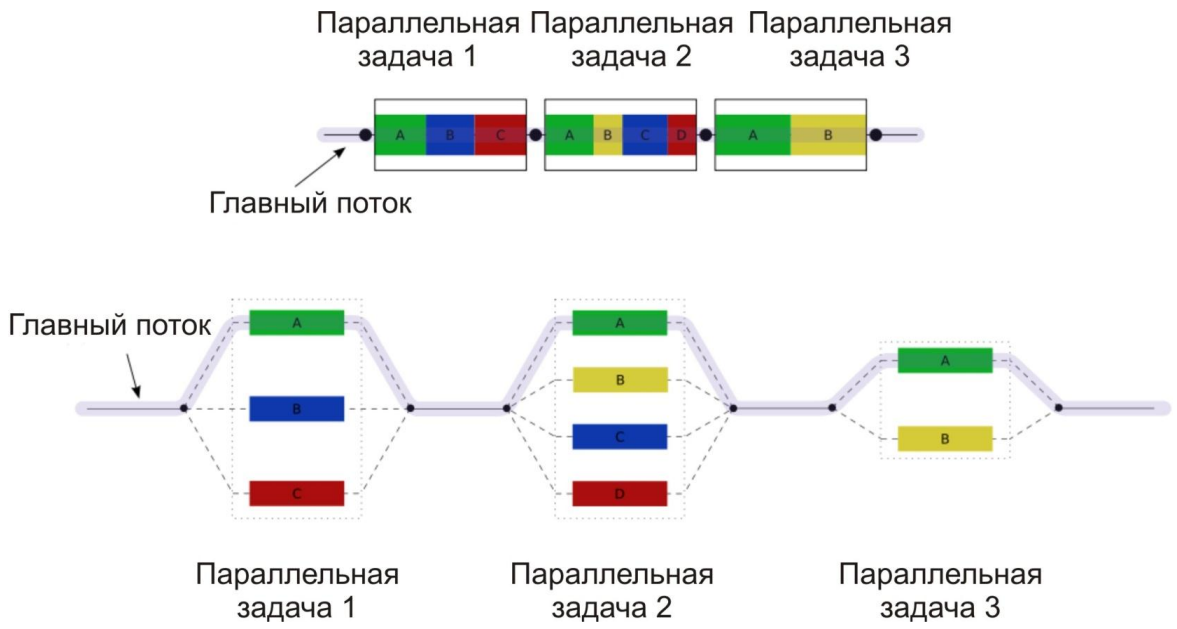


Рисунок 2 – Модель параллельной программы

OpenMP используется модель параллельного выполнения “ветвление-слияние” (fork-join). Программа начинается выполнением одной нити, называемой начальной (initial) нитью. Начальная нить выполняется последовательно. Когда нить достигает директивы parallel она создает команду нитей, состоящую из неё самой и нуля или более

дополнительных нитей, и становится хозяйкой (master) созданной команды. Все члены команды исполняют код структурной области, связанной с директивой parallel (параллельной области). В конце параллельной области размещается неявный барьер. Только нить-хозяйка продолжает выполнение после завершения параллельной области.

OpenMP прост в использовании и включает лишь два базовых типа конструкций: директивы pragma и функции исполняющей среды OpenMP. Директивы pragma, как правило, указывают компилятору, как реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с фразы pragma omp. Как и любые другие директивы pragma, они игнорируются компилятором, не поддерживающим конкретную технологию – в данном случае OpenMP. Каждая директива может иметь несколько дополнительных атрибутов. Отдельно специфицируются атрибуты для назначения классов переменных, которые могут быть атрибутами различных директив.

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

Число нитей в команде, выполняющихся параллельно, можно контролировать несколькими способами. Один из них – использование переменной окружения OMP_NUM_THREADS. Другой способ – вызов процедуры omp_set_num_threads(). Еще один способ – использование выражения num_threads в сочетании с директивой parallel.

На рисунке 3 приведен пример простой программы с использованием 2-х нитей и результат ее выполнения. На рисунке 4 – пример изменения числа нитей.

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

void main()
{
   omp_set_num_threads( 2 );
   #pragma omp parallel
   {
      cout << " Это одна строка " << endl;
   }
}
```

```
Выполнение

Это одна строка
Это одна строка
```

Рисунок 3 – Пример простой программы

```
void main()
{
omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
{
 cout<<"Параллельная область 1 \n";
}
#pragma omp parallel
{
 cout<<"Параллельная область 2 \n";
}
}
```

```
Результат
Параллельная область 1
Параллельная область 1
Параллельная область 1
Параллельная область 2
Параллельная область 2
```

Рисунок 4 – Изменение числа нитей

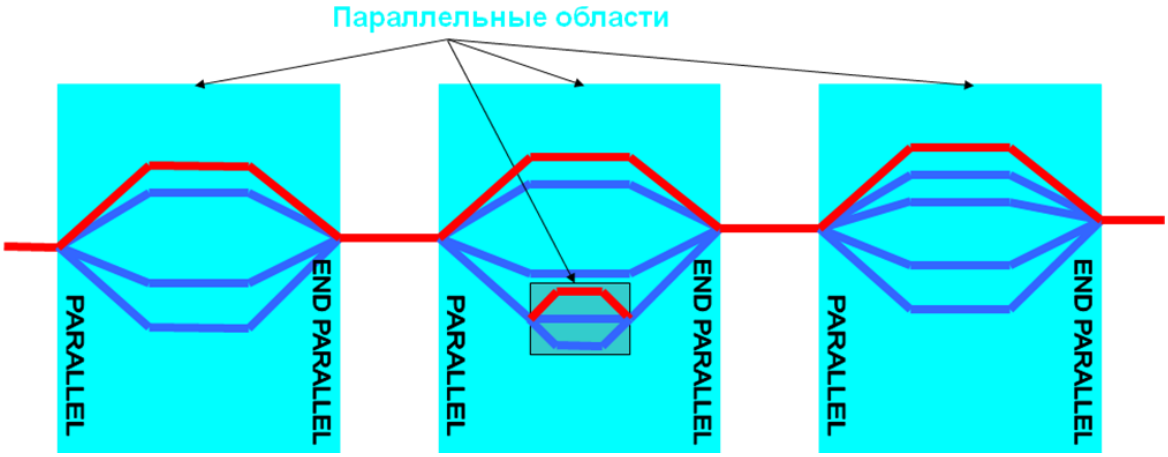
В программе может находится любое количество директив parallel (см. Рисунок 5). Параллельные области могут быть вложены друг в друга. Если вложенный параллелизм запрещен или не поддерживается реализацией, новая команда будет состоять только из самой нити, встретившей вложенную директиву parallel.

```
void main()
{
   const int n = 100;
   int a[n];
   omp_set_num_threads( 2 );
   #pragma omp parallel
   {
         #pragma omp for
         for (int i=0; i<n; i++)
            a[i]=i;
   }
}
```

Рисунок 5 – Разделение работы в параллельном регионе

Команда нитей, встретившая конструкцию распределения работы, разделяет работу, заданную этой конструкцией между нитями команды, и эта работа выполняется нитями совместно, вместо того чтобы выполняться полностью каждой нитью. После конструкции распределения работы все нити продолжают выполнение кода параллельной области.

Функция omp_set_nested(int) разрешает или запрещает вложенный параллелизм. Аргумент: 0 - выключение вложенного параллелизма, 1 – включение вложенного параллелизма (см. Рисунок 6).




```
int n;
omp_set_nested(1);
omp_set_num_threads(3);
#pragma omp parallel private(n)
{
n=omp_get_thread_num();
#pragma omp parallel
{
printf("Часть1, Поток %d - %d\n", n, omp_get_thread_num());
}
 }
omp_set_nested(0);
#pragma omp parallel private(n)
{
n=omp_get_thread_num();
#pragma omp parallel
{
printf("Часть 2, Поток %d - %d\n", n, omp_get_thread_num());
}
}
```

Рисунок 6 – Пример: Вложенные параллельные области

Конструкции синхронизации и библиотечные подпрограммы позволяют согласовывать выполнение нитей и доступ к данным в параллельных областях. Так как нет гарантий синхронного доступа к файлам при выполнении ввода/вывода из разных нитей, то синхронизация в этом случае возлагается на программиста.

Если в параллельной области какой-либо участок кода должен быть выполнен лишь один раз, то его нужно выделить директивами **single** (см. Рисунок 7).

```
void main()
{
omp_set_num_threads(3);
 #pragma omp parallel
 {
   cout<<"Сообщение 1"<<endl;
   #pragma omp single nowait
   {
    cout<<"Только один поток"<< endl;
   }
   cout<<«Сообщение 2"<<endl;
 }
}
```

Рисунок 7 – Директива single

Директива **master** выделяет участок кода, который будет выполнен только нитью-мастером (см. Рисунок 8).

```
void mode()
{ if(omp_in_parallel())   cout<<"Параллельный регион \n";
else   cout<<"Последовательный регион \n“;}
 void main()
{   omp_set_num_threads(2);
    mode();
#pragma omp parallel
 {
  #pragma omp master
    mode();
 }
}
```

```
Результат
Последовательный регион
Параллельный регион
```

Рисунок 8 – Директива master

OpenMP предполагает, что программа выполняется в системе с разделяемой памятью, в которой хранятся (могут быть сохранены и выбраны) переменные программы доступные всем её нитям. Кроме того каждая нить имеет доступ к памяти, недоступной для всех других нитей; такая память называется частной памятью нити.

Директива parallel разделяет память, доступную параллельной области, на разделяемую (shared) и частную (private) (см. Рисунок 9).

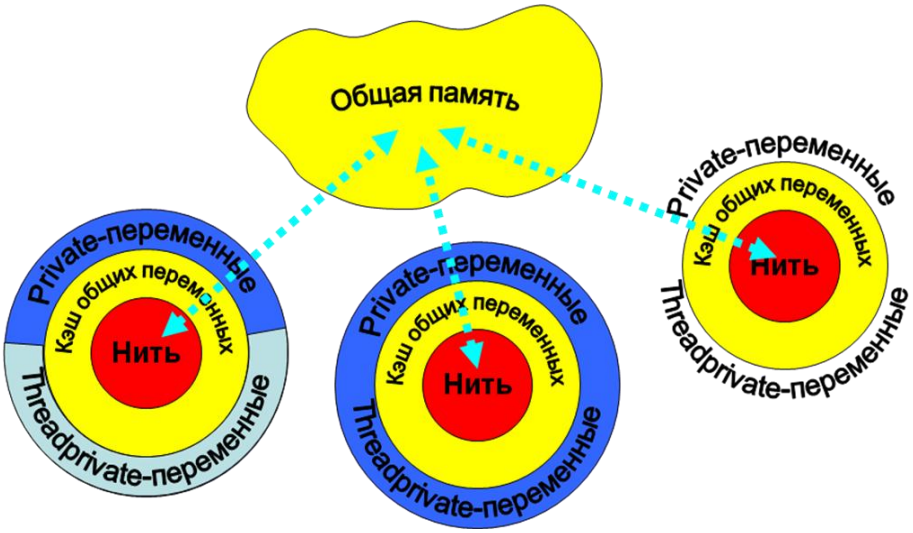


Рисунок 9 – Модель памяти OpenMP

Каждая переменная, встречающаяся в параллельной области, имеет соответствующую одноименную (оригинальную) переменную вне параллельной области. Разделяемая переменная ссылается на ту же память, что и оригинальная переменная вне параллельной области (пример – см. Рисунок 10). По умолчанию все переменные – разделяемые.

```
omp_set_num_threads( 3 );
int i = 1;
 cout<< " I = " << i<<endl;
 #pragma omp parallel share(i)
 {
        i = omp_get_thread_num();
        cout<<" i = "<<i<<endl;
 }
        cout<< " I = " << i<<endl;
```

```
Результат
I = 1
I = 2
I = 0
I = 1
I = 2
```

Рисунок 10 – Пример: разделяемые переменные

Для каждой частной переменной создается новая переменная в памяти каждой нити, кроме, быть может, нити-хозяйки. Частные переменные в параллельной области ссылаются на частную память нити (пример применения - см. Рисунок 11).

```
int f(int a)
{   return a*a-10*a-50; }
void main()
{
    const int n = 100;
    int a[n], t;
    omp_set_num_threads( 2 );
   #pragma omp parallel
   {
                    #pragma omp for private (t)
                     for (int i=0; i<n; i++)
                     {
                              t=f(i);
                              a[i]=t;
                     }
   }
}


 omp_set_num_threads( 3 );
 int i = 1;
  cout<< " I = " << i<<endl;
  #pragma omp parallel private(i)
  {
                 i = 5;
                 cout<< " I = " <<i<<endl;
  }
        cout<< " I = " << i<<endl;
```

Результат
I = 1
I = 5
I = 5
I = 5
I = 1

Рисунок 11 – Пример: использование клаузы private и локальные переменные

Дополнительно каждая нить может иметь своё временное представление памяти. Это временное представление не является необходимой частью модели памяти OpenMP, но учитывает наличие в

современных вычислительных системах регистров процессора, кэшей различных уровней и других структур, позволяющих нити кэшировать переменные и избегать лишних обращений к памяти. Это временное представление не обязано быть все время согласованным с памятью. Поэтому OpenMP предоставляет средства, позволяющие принудительно установить согласованность временного представления с памятью. Предполагается, что компилятор C/C++ поддерживающий OpenMP, автоматически поддерживает согласованность разделяемых переменных, описанных с ключевым словом volatile. Естественно предполагается, что для таких переменных проводится принудительное согласование непосредственно перед чтением и непосредственно после записи.

Опция **firstprivate()** задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере (см. Рисунок 12).

```
omp_set_num_threads( 2);
 int n=10;
 cout<<"Начальное значение n : "<<n<<endl;
#pragma omp parallel firstprivate(n)
{
cout<<" n = "<< n<<endl;
n=omp_get_thread_num();
cout<<"Новое значение n: "<< n<<endl;
}
cout<<"Последнее значение  n: "<< n<<endl;
```

Рисунок 12 – Использование опции firstprivate**()**

При использовании опции **copyprivate()**, после выполнения нити, содержащей конструкцию **single**, новые значения переменных списка будут доступны всем одноименным частным переменным, описанным в начале параллельной области и используемым всеми её нитями (см. Рисунок 13).

```
omp_set_num_threads(5);
int n;
#pragma omp parallel private(n)
{
n=omp_get_thread_num();
cout<<" Начальное n ="<<n<<endl;
#pragma omp single copyprivate(n)
{
n=100;
}
cout<<"Новое n = "<<n<<endl;
}
```

```
Результат
Начальное n = 2
Начальное n = 0
Начальное n = 1
Новое n = 100
Новое n = 100
Новое n = 100
```

Рисунок 13 – Использование опции copyprivate()

Опция **reduction(оператор:список)** задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор (пример применения опции – см. Рисунок 14). Список возможных операторов: **+, *, -, &, |, &&, ||** .

```
void main()
{
omp_set_num_threads(2);
int count = 0;
#pragma omp parallel reduction (+: count)
{
count++;
cout<<"Количество: "<< count<<endl;
}
cout<<"Число нитей: "<<count<<endl;
}
```

Рисунок 14 – Пример применения опции reduction(оператор:список)

Функция **omp_get_wtime()** возвращает в вызвавшей нити астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом (см. Рисунок 15).

```
double start_time, end_time;
int a[30];
omp_set_num_threads(2);
start_time = omp_get_wtime();
#pragma omp parallel
{
#pragma omp for
    for(int i = 0; i<30; i++)
      a[i]=j;
}
end_time = omp_get_wtime();
cout<<"Время на исполнение цикла"<< end_time-start_time;
```

Рисунок 15 – Функция для работы с системным таймером

Также для работы с системным таймером используется директива **threadprivate()**. Она может позволить сделать локальные копии для статических переменных языка Си, которые по умолчанию являются общими (см. Рисунок 16).

```
int n;
#pragma omp threadprivate(n)
void main()
{
n=5; int num;
omp_set_num_threads(2);
#pragma omp parallel private (num)
{
num=omp_get_thread_num();
cout<<"Значение n на нити " <<num<<" (на входе): "<<n<<endl;
n=omp_get_thread_num();
cout<<"Значение n на нити " <<num<<" (на выходе): "<<n<<endl;
}
cout<<"Значение n " <<n<<endl;
}
```

Результат
Значение n на нити 0 (на входе) 5
Значение n на нити 1 (на входе) 0
Значение n на нити 0 (на выходе) 0
Значение n на нити 1 (на выходе) 1
Значение n 0

Рисунок 16 – Использование статических переменных. Директива threadprivate()

Применение опции **copyin**() позволяет инициализировать локальные копии переменной n начальным значением нити-мастера (см. Рисунок 17).

```
int n;
#pragma omp threadprivate(n)
void main()
{
n=1; int num;
omp_set_num_threads(2);
#pragma omp parallel private (num) copyin(n)
{
num=omp_get_thread_num();
cout<<"Значение n на нити " <<num<<" (на входе): "<<n<<endl;
n=omp_get_thread_num();
cout<<"Значение n на нити " <<num<<" (на выходе): "<<n<<endl;
}
cout<<"Значение n " <<n<<endl;
}
```

```
Результат
Значение n на нити 0 (на входе) 5
Значение n на нити 1 (на входе) 5
Значение n на нити 0 (на выходе) 0
Значение n на нити 1 (на выходе) 1
Значение n 0
```

Рисунок 17 – Директива copyin()

Компиляторы не ограничиваются в переупорядочивании операций нити с памятью и использовании преимуществ временного представления, кроме необходимости сохранять порядок принудительных согласований. Поскольку согласование могут выполнять различные нити, OpenMP предполагает слабое упорядочение согласований.

Если в параллельной области встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву **for**.

#pragma omp for опция[[[,] опция] ...]

цикл for

Эта директива относится к идущему следом за данной директивой блоку, включающему оператор for.

Возможные опции:

- ✓ **private (список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;
- ✓ **firstprivate (список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;
- ✓ **lastprivate (список)** – переменным, перечисленным в списке, присваивается результат с последнего витка цикла;
- ✓ **reduction (оператор:список)** – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги); над локальными копиями переменных после завершения всех итераций цикла выполняется заданный оператор; оператор это: +, *, -, &, |, ^, &&, ||; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;
- ✓ **schedule(type[, chunk])** – опция задаёт, каким образом итерации цикла распределяются между нитями;
- ✓ **collapse(n)** – опция указывает, что n последовательных тесновложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция collapse не задана, то директива относится только к одному непосредственно следующему за ней циклу;
- ✓ **ordered** – опция, говорящая о том, что в цикле могут встречаться директивы ordered; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;
- ✓ **nowait** – в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их

дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция nowait позволяет нитям, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными.

На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит. Нельзя использовать побочный выход из параллельного цикла. Размер блока итераций, указанный в опции schedule, не должен изменяться в рамках цикла.

Эти требования введены для того, чтобы OpenMP мог при входе в цикл точно определить число итераций. Если директива параллельного выполнения стоит перед гнездом циклов, завершающихся одним оператором, то директива действует только на самый внешний цикл. Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции lastprivate.

Диспетчеризация циклов осуществляется при помощи опции **schedule(type[, chunk])**, которая задаёт, каким образом итерации цикла распределяются между нитями (пример применения – см. Рисунок 18).

В опции **schedule** параметр **type** задаёт следующий тип распределения итераций:

- ✓ **static** – блочно-циклическое распределение итераций цикла; размер блока – chunk. Первый блок из chunk итераций выполняет нулевая нить, второй блок — следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити (см. Рисунок 19).

```
omp_set_num_threads(3);
int A[100], B[100], C[100], i, n;

for (i=0; i<100; i++){ A[i]=i; B[i]=2*i; C[i]=0; }

#pragma omp parallel private(i, n)
{
n=omp_get_thread_num();
#pragma omp for schedule(dynamic,5)
for (i=0; i<100; i++)
{
C[i]=A[i]+B[i];
cout<<"Нить "<< n<< " сложила элементы с номером " <<i<<endl;
}
}
```

i	static	static, 1	static, 2	dynamic	dynamic, 2	guided	guided, 2
0	0	0	0	0	0	0	0
1	0	1	0	1	0	2	0
2	0	2	1	2	1	1	1
3	1	3	1	3	1	3	1
4	1	0	2	1	2	1	2
5	1	1	2	3	2	2	2
6	2	2	3	2	3	3	3
7	2	3	3	0	3	0	3
8	3	0	0	1	3	0	0
9	3	1	0	0	3	3	0

Рисунок 18 – Пример: Диспетчеризация циклов

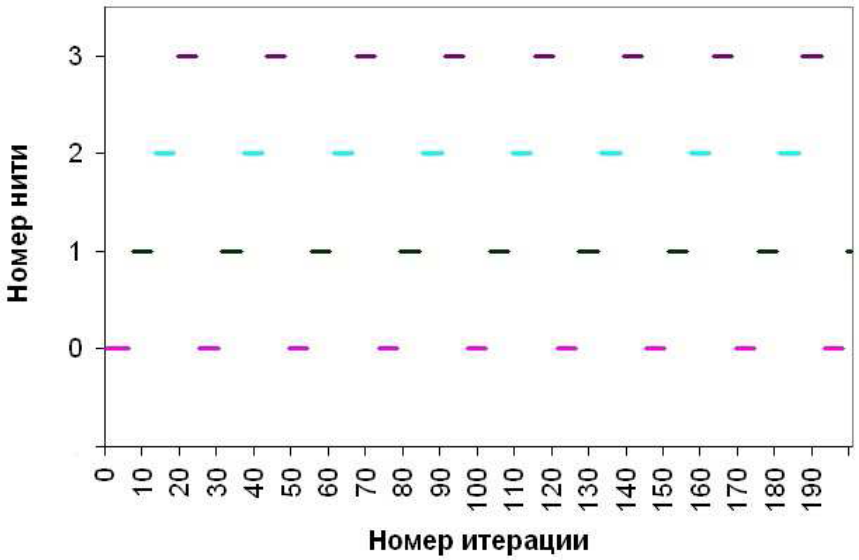


Рисунок 19 – Распределение итераций по нитям для static(6)

- ✓ **dynamic** – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает chunk итераций (по умолчанию chunk=1), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из chunk итераций (см. Рисунок 20).
- ✓ **guided** – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины chunk (по умолчанию chunk=1) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл (см. Рисунок 21).
- ✓ **runtime** – способ распределения итераций выбирается во время работы программы по значению переменной среды OMP_SCHEDULE. Параметр chunk при этом не задаётся.

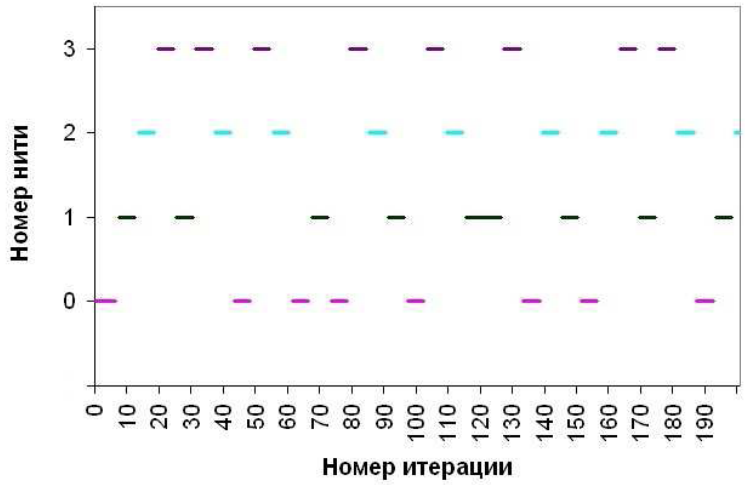


Рисунок 20 – Распределение итераций по нитям для dynamic(6)

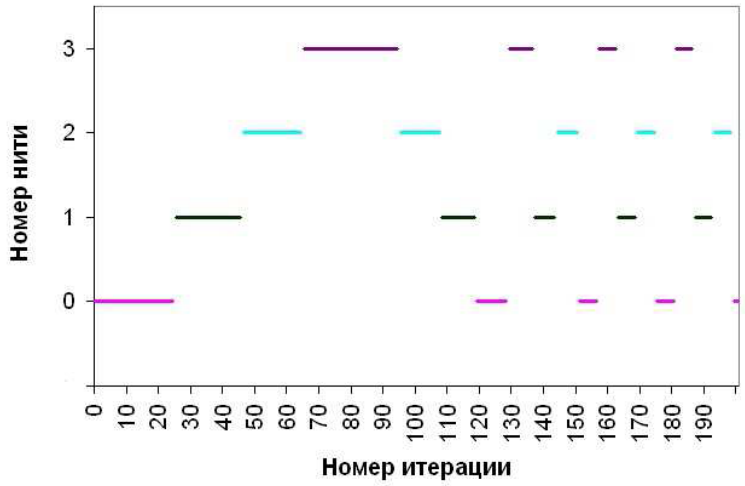


Рисунок 21 – Распределение итераций по нитям для guided(6)

Для использования механизмов OpenMP нужно скомпилировать программу компилятором, поддерживающим OpenMP, с указанием соответствующего ключа, например:

icc/ifort используется ключ компилятора -openmp

gcc /gfortran -fopenmp

Sun Studio -xopenmp

Visual C++ - /openmp

PGI -mp

Компилятор интерпретирует директивы OpenMP и создаёт параллельный код. При использовании компиляторов, не поддерживающих OpenMP, директивы OpenMP игнорируются без дополнительных сообщений. Компилятор с поддержкой OpenMP определяет макрос _OPENMP, который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы. При необходимости это определение можно протестировать, как показано ниже:

```
#ifdef _OPENMP
        fn();
#endif
```

2 Практические занятия

2.1 Практическое занятие 1. Начало работы с OpenMP

2.1.1 Создание проекта в среде MS Visual Studio с поддержкой OpenMP

1. Запустите Microsoft Visual Studio 2013. При первом запуске Visual Studio выберите интерфейс по умолчанию "Параметры разработки Visual C++".

2. Создайте новый проект.

3. В окне "Создать проект" в раскрывающемся списке выберите "Visual C++". В подокне в середине выберите "Консольное приложение Win32". Внизу введите имя проекта (например, Example) и место расположения проекта, нажмите кнопку OK (см. Рисунок 28).

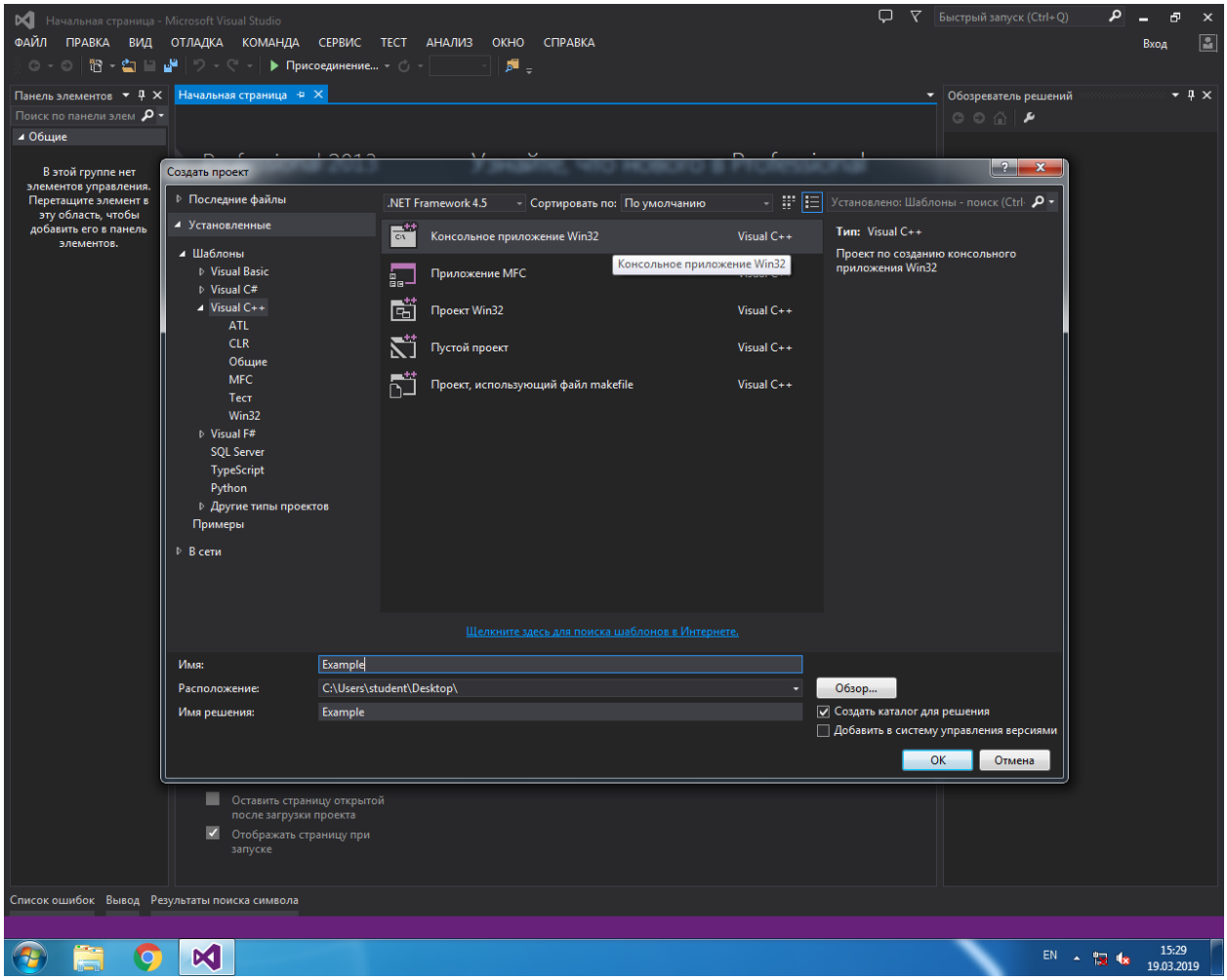


Рисунок 28 – Создание проекта в среде MS Visual Studio

4. В открывшемся окне "Обозреватель решений – Решение "Example" (см. Рисунок 29) для включения поддержки OpenMP установите дополнительные параметры компиляции проекта: нажатием правой клавиши мыши на имени "Example" вызовите всплывающее окно, в котором выберите пункт "Свойства" (см. Рисунок 30).

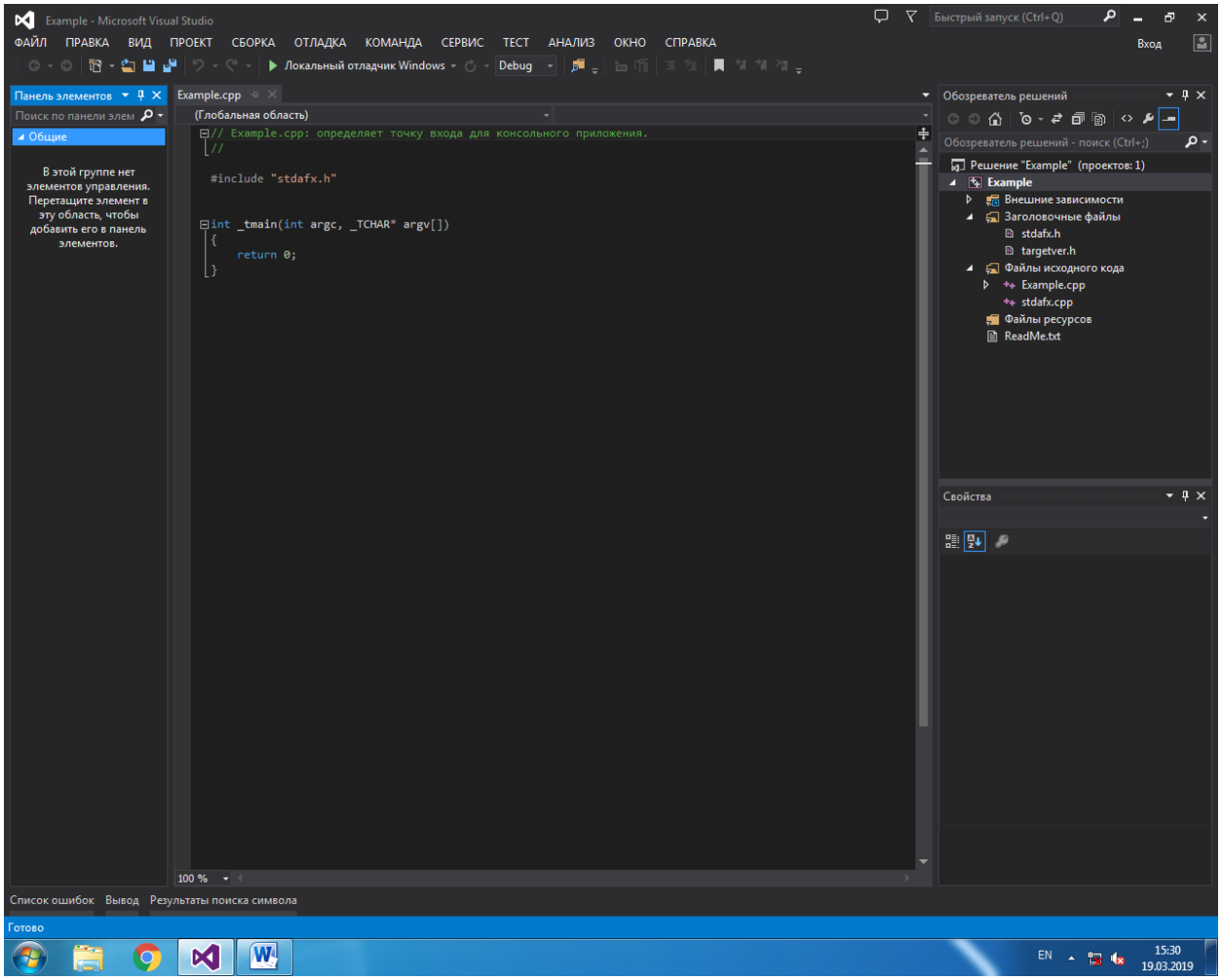


Рисунок 29 – Окно "Обозреватель решений – Решение "Example"

5. В открывшемся окне выберите "Свойства конфигурации / C/C++ / Язык". Установите для опции "Поддержка OpenMP" значение "Да (/openmp)". Нажмите кнопку OK (см. Рисунок 31).

6. Сохраните проект.

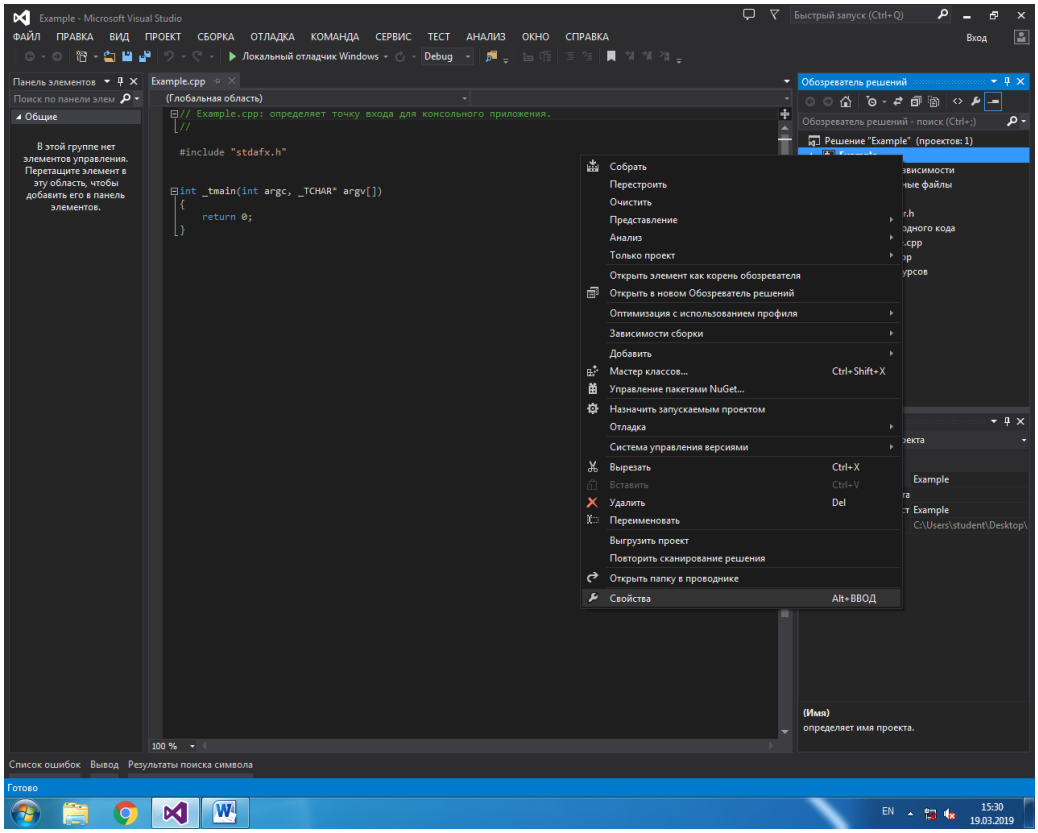


Рисунок 30 – Вызов свойств решения "Example"

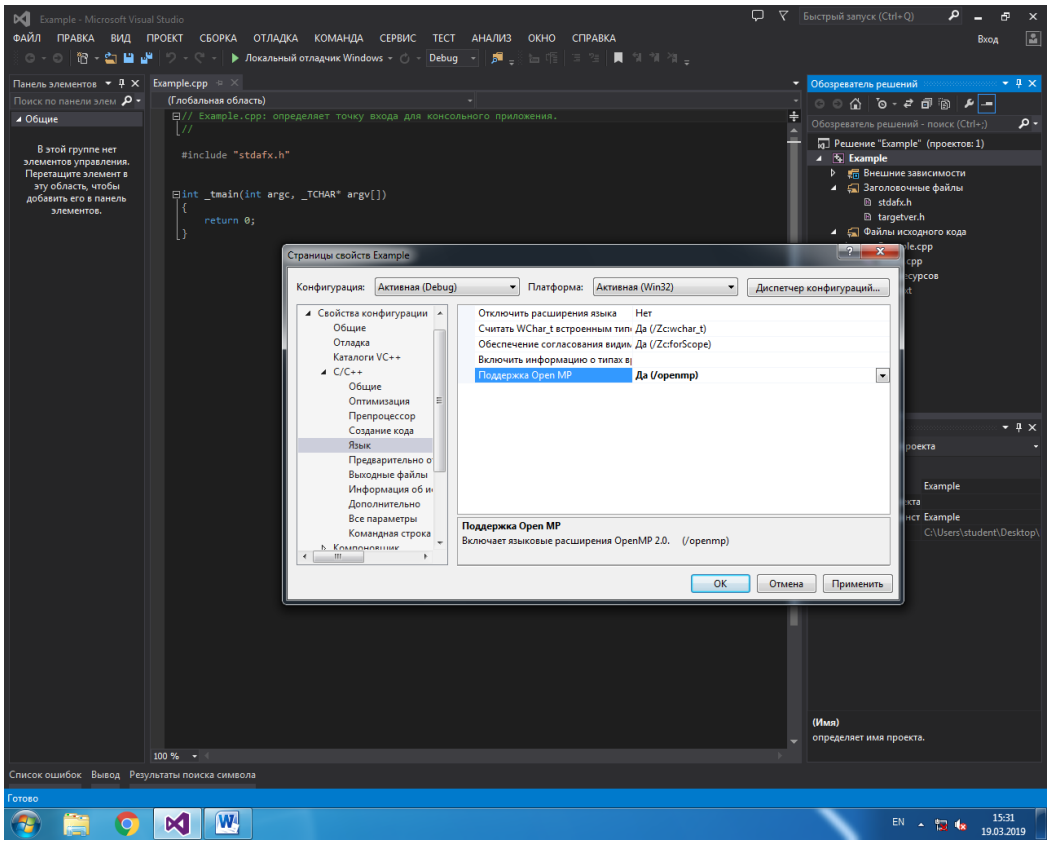


Рисунок 31 – Включение поддержки OpenMP

2.1.2 Знакомство с результатами выполнения типовых задач

Наберите, откомпилируйте и запустите приведенные ниже типовые программы, ознакомьтесь с результатами их выполнения.

2.1.2.1 Установка числа потоков

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

using namespace std;

void main()
{
	omp_set_num_threads( 2 ); //Установка числа потоков
	#pragma omp parallel
	{
		cout << "This is a line " << endl;
	}
}
```

2.1.2.2 Получение числа потоков

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

using namespace std;

void main()
{
	omp_set_num_threads( 2 );
	#pragma omp parallel
	{
		cout <<"The number of the threads:
	"<<omp_get_num_threads()<<endl;
	}
}
```

2.1.2.3 Параллельное заполнение массива

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

void main()
```

```
{
	const int n = 100;
	int a[n];

	omp_set_num_threads( 2 );
	pragma omp parallel
	{
		#pragma omp for
		for (int i=0; i<n; i++)
			a[i]=i;
	}
}
```

2.1.2.4 Определение времени выполнения параллельного региона

Функция omp_get_wtime() позволяет получать астрономическое время в секундах. Представленный ниже код демонстрирует её использование для получения времени выполнения параллельного региона.

```
#include <iostream>
#include <omp.h>
using namespace std;

void main()
{
	double start_time, end_time;
	int a[30];
	omp_set_num_threads(2);
	start_time = omp_get_wtime();
	#pragma omp parallel
	{
		#pragma omp for
		for(int i = 0; i<30; i++)
			a[i]=i;
	}
	end_time = omp_get_wtime();
	cout<<"Время на исполнение цикла"<< end_time-start_time;
}
```

2.1.3 Задание 1. Решить задачу определения времени заполнения массива

Необходимо написать программу, которая на осуществляет заполнение массива *a* из 20 элементов по правилу *a[i]=0.5*i*i-8.5*i+0.3* главной нитью (до входа в параллельный регион) и массива b из 20 элементов по такому же правилу *b[i]=0.5*i*i-8.5*i+0.3* но уже в параллельной области для двух потоков. Оценить время заполнения массивов и вывести на экран имя того массива и все его элементы, который заполнился быстрее.

Посмотреть, как изменилось время выполнения программы если массивы содержат 100, 1000, 50000, 100000 элементов.

2.2 Практическое занятие 2. Потоки

2.2.1 Знакомство с результатами выполнения типовых задач

2.2.1.1 Получение идентификатора потока

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

using namespace std;

void main()
{
	int mId, nthreads;
	omp_set_num_threads( 2 );
	#pragma omp parallel private(mId, nthreads)
	{
		mId = omp_get_thread_num();
		nthreads=omp_get_num_threads();
		cout<<"ID = "<<mId<<" The number of the threads =
	"<<nthreads<<endl;
		if (mId==0)
			cout<<"This is the Master Thread"<<endl;
		else
			cout<<"This is another thread"<<endl;
	}
}
```

2.2.1.2 Изменение числа нитей

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

using namespace std;

void main()
{
	omp_set_num_threads(2);
	#pragma omp parallel num_threads(3)
	{
		cout<<"Parallel region 1 \n";
	}
	#pragma omp parallel
	{
		cout<<"Parallel region 2 \n";
	}
}
```

2.2.1.3 Последовательный или параллельный регион. Использование МАСТЕР-потока

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

using namespace std;
void mode()
{
	if(omp_in_parallel())
		cout<<"Parallel region \n";
	else
		cout<<"Concurrent region \n";
}

void main()
{
	omp_set_num_threads(5);
	mode();
	#pragma omp parallel
	{
		#pragma omp master
```

```
            {
                mode();
            }
        }
}
```

2.2.1.4 Использование локальных переменных

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

using namespace std;
int f(int a)
{
        return a*a-10*a-50;
}
void main()
{
        const int n = 100;
        int a[n], t;

        omp_set_num_threads( 2 );
        #pragma omp parallel
        {
            #pragma omp for private (t)
            for (int i=0; i<n; i++)
            {
                t=f(i);
                a[i]=t;
            }
        }
}
```

2.2.2 Задание 2.1. Решить задачу запуска программы на всех доступных процессорах и определения номера потока

Функция omp_get_num_procs() позволяет получить общее количество процессоров доступных системе. Необходимо добавить в программу № 3.2.1.4 "Использование локальных переменных" возможность запускать эту программу на всех доступных процессорах и выводить на экран те элементы (а также их количество), которые были заполнены главным потоком.

2.2.3 Задание 2.2. Решить задачу изменения вложенных потоков

Изменить программу, данную ниже, так чтобы вложенная параллельная область выполнялся только а) одним потоком, б) главной нитью, в) главной нитью и число вложенных потоков было равно 10.

```
#include <iostream>
#include <omp.h>
using namespace std;

void main()
{
	int n;
	omp_set_nested(1);
	omp_set_num_threads(5);
	#pragma omp parallel private(n)
	{
		n=omp_get_thread_num();
		#pragma omp parallel
		{
			printf("Часть 1 Поток %d - %d \n",n,
			omp_get_thread_num());
		}
	}
	omp_set_nested(0);
	#pragma omp parallel private(n)
	{
		n=omp_get_thread_num();
		#pragma omp parallel
		{
			printf("Часть 2 Поток %d - %d \n",n,
		omp_get_thread_num());
		}
	}
}
```

2.3 Практическое занятие 3. Разделяемые переменные. Синхронизация потоков

2.3.1 Знакомство с результатами выполнения типовых задач

2.3.1.1 Использование разделяемых переменных

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

using namespace std;

void main()
{
	int i = 5;
	omp_set_num_threads( 10 );
	#pragma omp parallel shared(i)
	{
		i = omp_get_thread_num();
		cout<<" i = "<<i<<endl;
	}
	cout<<"i = "<<i;
}
```

2.3.1.2 Однократное выполнение/ Барьерная синхронизация

```
#include <iostream>
#include <omp.h>
using namespace std;
void main()
{
	omp_set_num_threads(5);
	#pragma omp parallel
	{
		cout<<"Message 1"<<endl;
		#pragma omp single nowait
		{
			cout<<"One thread"<<endl;
		}
		cout<<"Message 2"<<endl;
	}
}
```

2.3.1.3 Использование редукции

```
#include <stdio.h>
#include <omp.h>
using namespace std;

void main()
{
	int count = 0 ;

	omp_set_num_threads( 2 );
	#pragma omp parallel reduction(+:count)
	{
		count++;
	}
	printf("Число нитей равно %d", count);
}
```

2.3.1.4 Использование переменных threadprivate

```
#include <stdio.h>
#include <omp.h>
using namespace std;
int n;
#pragma omp threadprivate(n)

int main()
{
	int num;
	n=1;
	omp_set_num_threads(3);
	#pragma omp parallel private (num)// copyin(n)
	{
		num=omp_get_thread_num();
		printf("Value n thread %d (enter): %d \n",num, n);
		n=omp_get_thread_num();
		printf("Value n thread %d (exit): %d \n",num, n);
	}
	printf("Value n = %d \n",n);

	#pragma omp parallel private (num) //num_threads(4)
	{
```

```
		num=omp_get_thread_num();
		printf("Value n thread %d = %d \n",num, n);
	}
}
```

2.3.1.5 Использование барьера

```
#include <omp.h>
#include <stdio.h>
using namespace std;

int main()
{
	int num;
	omp_set_num_threads(4);
	#pragma omp parallel private(num)
	{
		num=omp_get_thread_num();
		printf("Parallel region 1 Value ID= %d \n",num);
		#pragma omp barrier
		printf("Parallel region 2 Value ID= %d \n",num);
	}
}
```

2.3.1.6 Использование atomic

```
#include <omp.h>
#include <iostream>
#include <stdio.h>
using namespace std;

int main()
{
	int num;
	omp_set_num_threads(4);
	int count = 0;
	#pragma omp parallel
	{
		#pragma omp atomic
		count++;
	}
	printf("Число нитей: %d\n", count);
}
```

2.3.1.7 Использование copyprivate

```
#include <iostream>
#include <omp.h>
using namespace std;

void main()
{
	omp_set_num_threads(5);
	int n;
	#pragma omp parallel private(n)
	{
		n=omp_get_thread_num();
		cout<<" Start n ="<<n<<endl;
		#pragma omp single copyprivate(n)
		{
			n=100;
		}
		cout<<"END n = "<<n<<endl;
	}
}
```

2.3.2 Задание 3. Решить задачу формирования локальной копии вектора

Имеется вектор *a* из 40 элементов. Первую половину этого вектора нужно заполнить случайными числами по правилу: a[i]= rand()%100. Параллельная область запускается на всех доступных процессорах. Для каждого потока сформировать локальную копию вектора, оставшуюся половину вектора проинициализировать номером потока. Распечатать все вектора и номера потоков, в которых они хранятся.

2.4 Практическое занятие 4. Циклические операции

2.4.1 Знакомство с результатами выполнения типовых задач

2.4.1.1 Использование директивы for

В последовательной области инициализируются три исходных массива A, B, C. В параллельной области данные массивы объявлены общими. Вспомогательные переменные i и n объявлены локальными.

Каждая нить присвоит переменной n свой порядковый номер. Далее с помощью директивы for определяется цикл, итерации которого будут распределены между существующими нитями. На каждой i-ой итерации данный цикл сложит i-ые элементы массивов A и B и результат запишет в i-ый элемент массива C. Также на каждой итерации будет напечатан номер нити, выполнившей данную итерацию.

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

int main(int argc, char *argv[])
{
	int A[10], B[10], C[10], i, n;
	// Заполним исходные массивы
	for (i = 0; i < 10; i++)
	{
		A[i] = i;
		B[i] = 2 * i;
		C[i] = 0;
	}
	#pragma omp parallel shared(A, B, C) private(i, n)
	{
		// Получим номер текущей нити
		n = omp_get_thread_num();
		#pragma omp for
			for (i = 0; i < 10; i++)
			{
				C[i] = A[i] + B[i];
				printf("Нить \%d сложила элементы с номером
				%d\n", n, i);
			}
	}
}
```

2.4.1.2 Использование опции ordered

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

```
using namespace std;

int main()
{
	int num;
	omp_set_num_threads(4);
	int i, n;
	#pragma omp parallel private (i, n)
	{
		n=omp_get_thread_num();
		#pragma omp for ordered
		for (i=0; i<5; i++)
		{
			printf("Thread %d, Iteration %d\n", n, i);
			#pragma omp ordered
			{
				printf("ordered: Thread %d, Iteration %d\n", n, i);
			}
		}
	}
}
```

2.4.2 Задание 4.1. Решить задачу вычисление произведения квадратных матриц

Матрица a есть произведение матриц b и c, если

$$a_{ij} = \sum_{k=1}^{n} b_{ik} * c_{kj}.$$

Требуется написать программу, производящую вычисление произведения квадратных матриц *a* и *b* размером 100x100. Оценить время выполнения. Матрицы инициализируются по правилу: a[i][j]=b[i][j]=i*j;

2.4.3 Задание 4.2. Решить задачу сложения двух массивов при нескольких режимах диспетчеризации

Необходимо написать программу и посмотреть работу программы сложения двух массивов (вывести элементы массива, и номер потока, посчитавший найденный элемент) при нескольких режимах диспетчеризации (guided,static, dynamic). Число потоков установить равным числу вычислительных ядер. Размерность вектора – 100 элементов.

2.4.4 Задание 4.3. Решить задачу вычисления и вывода значений элементов массива

Имеется двумерный массив чисел (значения производные) a с размерами 5x5. Используя параллельный цикл, сформировать массив b по следующему правилу $b[i] = \sum_{j=1}^{5} a_{ij}$.

В этом же цикле вывести:

а) значения ***b[i]*** в порядке их вычисления *и* номера потоков, вычисливших эти элементы ***b[i]***.

б) значения ***b[i]*** в порядке возрастания индекса ***i*** *и* номера потоков, вычисливших эти элементы ***b[i]***.

2.5 Практическое занятие 5. Секции

2.5.1 Знакомство с результатами выполнения типовых задач

2.5.1.1 Параллельные секции

```
#include <stdio.h>
#include <omp.h>
using namespace std;

void main()
{
	int num;
	omp_set_num_threads(3);
	#pragma omp parallel private(num)
	{
		num=omp_get_thread_num();
		#pragma omp sections
		{
			#pragma omp section
			printf("First section ID= %d\n", num);
			#pragma omp section
			printf("Second section ID= %d\n",num);
			#pragma omp section
			printf("Third section ID= %d\n", num);
		}
		printf("Parralel region \n");
	}
}
```

2.5.1.2 Использование опции lastprivate

```
#include <stdio.h>
#include <omp.h>
using namespace std;
void main()
{
	int num, n;
	omp_set_num_threads(3);
	#pragma omp parallel private(num)
	{
		num=omp_get_thread_num();
		#pragma omp sections lastprivate (n)
		{
			#pragma omp section
			{
				n = 1;
				printf("First section ID= %d n = %d\n", num, n);
			}
			#pragma omp section
			{
				n = 2;
				printf("Second section ID= %d n = %d\n",num, n);
			}
			#pragma omp section
			{
				n = 3;
				printf("Third section ID= %d n = %d\n", num, n);
			}
		}
		printf("Parralel region ID = %d n = %d\n", num, n);
	}
}
```

2.5.1.3 Использование критических секций

```
#include <omp.h>
#include <stdio.h>
using namespace std;
int main()
{
	int num;
```

```
	omp_set_num_threads(16);
	int i, n;
	#pragma omp parallel
	{
		#pragma omp critical
		{
			n=omp_get_thread_num();
			printf("Thread %d\n", n);
		}
	}
}
```

2.5.2 Задание 5. Решить задачу с использованием директив sections и critical

Имеются две функции: ***f1(x)=x*sin(x)-2*cos(x)*** ***и*** ***f2(x)=2*sin(x)-x*cos(x)***. Определить, сколько из этих функций при ***x=0.5*** попадает в интервал (-1; 1). Вычисления производить на двух потоках, используя директивы sections и critical.

2.6 Практическое занятие 6. Замки

2.6.1 Знакомство с результатами выполнения типовых задач

2.6.1.1 Использование замков 1

```
#include <omp.h>
#include <iostream>
#include <stdio.h>
#include <windows.h>
using namespace std;
int main()
{
	int num;
	omp_lock_t lock;
	int n;
	omp_set_num_threads(3);
	omp_init_lock(&lock);
	#pragma omp parallel private (n)
	{
		n=omp_get_thread_num();
		omp_set_lock(&lock);
```

```
            printf("Start of closed section, Thread %d\n", n);
            Sleep(5);
            printf("End of closed section, Thread %d\n", n);
        omp_unset_lock(&lock);
        }
        omp_destroy_lock(&lock);
}
```

2.6.1.2 Использование замков 2

```
#include <omp.h>
#include <stdio.h>
#include <windows.h>
using namespace std;
int main()
{
        int num;
        omp_lock_t lock;
        int n;
        omp_set_num_threads(3);
        omp_init_lock(&lock);
        #pragma omp parallel private (n)
        {
            n=omp_get_thread_num();
            while (!omp_test_lock (&lock))
            {
                printf("Section is closed, Thread %d\n", n);
                Sleep(2);
            }
            printf("Start of closed section, Thread %d\n", n);
            Sleep(5);
            printf("End of closed section, Tread %d\n", n);
            omp_unset_lock(&lock);
        }
        omp_destroy_lock(&lock);
}
```

2.6.1 Задание 6. Решить задачу с использованием простых замков

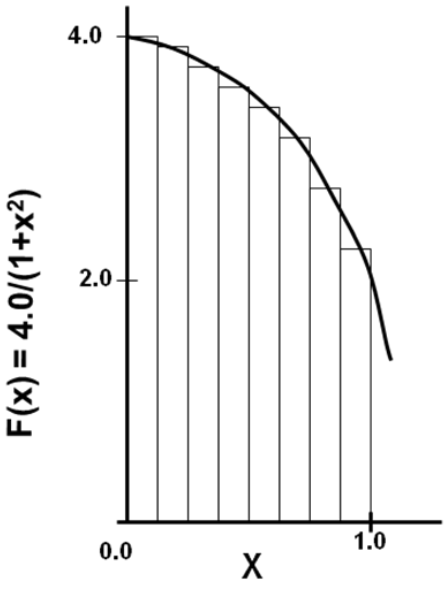
Решить задачу 3.5.2, используя простые замки вместо critical.

3 Индивидуальные задания

Целью выполнения индивидуального задания является изучение возможности и эффективности организации параллельных вычислений для выбранного алгоритма, разработка программной реализации его последовательной и параллельной версий.

3.1 Пример выполнения индивидуального задания - вычисление числа π

Последовательная реализация алгоритма вычисления числа π, представленного на Рисунке 25, показана на Рисунке 26, а его параллельная реализация с помощью технологии OpenMP – на рисунке 27.



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

Мы можем аппроксимировать интеграл как сумму прямоугольников:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi,$$

где каждый прямоугольник имеет ширину Δx и высоту $F(x_i)$ в середине интервала

Рисунок 25 – Вычисление числа π

```
#include <iostream.h>
void main ()
{
   int n =100000, i;
   double pi, h, sum, x;
   h   = 1.0 / (double) n;
   sum = 0.0;
   for (i = 1; i <= n; i ++)
   {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
    }
    pi = h * sum;
    cout<<"pi ="<< pi);
}
```

Рисунок 26 – Вычисление числа π (последовательная реализация)

```
#include <iostream.h>
void main ()
{
   int n =100000, i;
   double pi, h, sum, x;
   h   = 1.0 / (double) n;
   sum = 0.0;
#pragma omp parallel for reduction(+:sum) private(x)
   for (i = 1; i <= n; i ++)
   {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
    }
    pi = h * sum;
   cout<<"pi ="<< pi);
}
```

Рисунок 27 – Вычисление числа π (реализация с помощью OpenMP)

3.2 Варианты индивидуальных заданий

1. Вычисление скалярного произведения векторов
2. Умножение матрицы на вектор
3. Умножение матриц
4. Пузырьковая сортировка
5. Чет-нечетная перестановка
6. Сортировка слиянием

7. Сортировка Шелла
8. Сортировка Бэтчера
9. Базовая быстрая сортировка
10. Улучшенная быстрая сортировка
11. Вычисление корней алгебраического уравнения
12. Вычисление корней трансцендентного уравнения
13. Решение системы алгебраических уравнений методом простой итерации
14. Решение системы линейных уравнений методом Гаусса
15. Решение системы линейных уравнений методом сопряженных градиентов
16. Умножение матриц при ленточной схеме разделения данных
17. Умножение матриц при блочной схеме разделения данных
18. Алгоритм Фокса умножения матриц
19. Алгоритм Кэннона умножения матриц
20. Задача поиска всех кратчайших путей графа
21. Задача оптимального разделения графов

4 Методические указания к выполнению практических заданий

Перед выполнением заданий необходимо изучить теоретический материал по теме занятия.

Для ознакомления с результатами работы типовых программ следует набрать приведенные коды, произвести их компиляцию и запустить на исполнение. Проанализировать полученные результаты.

На каждом практическом занятии необходимо выполнить все приведенные задания, результаты оформить в виде отчета.

Индивидуальное задание выполняется в следующей последовательности:

- разрабатывается алгоритм решения выбранной задачи;
- разрабатывается последовательная реализация этого алгоритма, содержащая в себе оценку времени исполнения программного кода;
- разрабатывается параллельная реализация алгоритма решения задачи, также с оценкой времени исполнения;
- производится сравнение времен исполнения последовательной и параллельной реализаций, делаются выводы о возможности и эффективности применения технологии OpenMP для решения выбранной задачи.

5 Требования к содержанию и оформлению отчета о выполнении практических заданий

Содержание отчета о выполнении практических заданий:

1. Решение задач по темам практических занятий:
   - 1.1. Формулировка задачи.
   - 1.2. Описание программной реализации (код программы с комментариями).
   - 1.3. Пример результата выполнения программы (PrintScreen окна результатов).
2. Применение методов параллельных вычислений для реализации алгоритма (название алгоритма выбранного индивидуального задания)
   - 2.1. Формулировка задания.
   - 2.2. Блок-схема алгоритма решения задачи.
   - 2.3. Описание последовательной программной реализации алгоритма (код программы с комментариями);
   - 2.3. Пример результата выполнения последовательной реализации (PrintScreen окна результатов).
   - 2.4. Описание программной реализации алгоритма с использованием технологии OpenMP (код программы с комментариями);
   - 2.5. Пример результата выполнения параллельной реализации (PrintScreen окна результатов);
   - 2.6. Выводы о возможности и эффективности применения технологии OpenMP для решения выбранной задачи.

Отчет о выполнении практических заданий оформляется в соответствии с требованиями учебно-методического пособия «Правила оформления учебных работ студентов» [5].

Оформленную работу необходимо сдать в электронном виде, включая рабочие варианты программ (можно в виде исполняемых файлов).

Список рекомендуемой литературы

1. **Гергель, В.П.** Теория и практика параллельных вычислений / В.П. Гергель. - М. : Интернет-Университет Информационных Технологий, 2007. - 424 с. - (Основы информационных технологий). - ISBN 978-5-9556-0096-3 ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=233067

2. **Левин, М.П.** Параллельное программирование с использованием OpenMP / М.П. Левин. - М. : Интернет-Университет Информационных Технологий, 2008. - 120 с. - (Основы информационных технологий). - ISBN 978-5-94774-857-4 ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=233111.

3. **Антонов, А.С.** Параллельное программирование с использованием технологии OpenMP [Текст]: Учебное пособие. / А.С. Антонов – М.: Изд-во МГУ, 2009. - 77 с.

4. **Лупин, С.А.** Технологии параллельного программирования [Текст]: Учебное пособие / С.А. Лупин, М.А. Посыпкин. – М.: ИД «ФОРУМ»: ИНФРА-М, 20011. - 208 с. – (Высшее образование)

5. **Жибинова, И. А.** Правила оформления учебных работ студентов [Текст] : учебно-методическое пособие / И. А. Жибинова [и др.]; Новокузнец. ин-т (фил.) Кемеров. гос. ун-та ; под ред. И. А. Жибиновой. – Новокузнецк: НФИ КемГУ, 2018. – 124 с.

Приложение А

Образец титульного листа к отчету о выполнении практических заданий

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Новокузнецкий институт (филиал)
федерального государственного бюджетного образовательного
учреждения высшего образования
«Кемеровский государственный университет»

Факультет информатики, математики и экономики

Кафедра информатики и вычислительной техники им. В. К. Буторина

Иванов Иван Иванович
гр. ИВТ-16-1

ОТЧЕТ О ВЫПОЛНЕНИИ ПРАКТИЧЕСКИХ ЗАДАНИЙ

по дисциплине «Технологии параллельного программирования»

по направлению подготовки 09.03.01 Информатика и вычислительная техника
направленность (профиль) подготовки «Автоматизированные системы обработки информации и управления»

Проверил:
канд. техн. наук
О.В. Михайлова

Общий балл:_____

Оценка: _____

_____
подпись

«____» _____ 20____ г.

Новокузнецк 20_____