

Лабораторная работа №11. Разработка параллельных программ с применением технологии OpenMP

Оглавление

ВВЕДЕНИЕ	1
1. Модель параллельной программы OpenMP	2
2. Директивы и функции OpenMP	3
3. Простейшая OpenMP программа.....	4
3.1. Описание программы	4
3.2. Компиляция и выполнение OpenMP программы	4
4. Параллельные секции, переменные среды и замер времени	5
5. Варианты распределения работы между нитями параллельной программы	7
5.1. Низкоуровневое программирование.....	7
5.2. Распараллеливание оператора цикла.....	8
6. Распараллеливание линейной программы директивами OpenMP	10
7. Примеры программ с использованием технологии OpenMP	12
7.1. Параллельное перемножение двух квадратных матриц	12
7.2. Приближенное вычисление определенного интеграла.....	12
7.3. Использование OpenMP в MPI программе	13
ЗАДАНИЕ	14

ВВЕДЕНИЕ

Решая различные задачи компьютерного моделирования на суперкомпьютере, мы познакомились с технологией MPI описания параллельных потоков и информационного взаимодействия таких потоков. Даже на том уровне, который был нам доступен в рамках этого краткого курса, можно было выделить следующие основные характеристики технологии MPI:

1. Количество задействованных потоков определяется при запуске параллельной программы (команда `mpirun`, ключ `-np`). Это количество не может быть изменено средствами MPI.
2. Каждый поток работает с одной и той же копией параллельной программы (поэтому в параллельном режиме можно запустить и обычную линейную программу).

3. Копия параллельной программы, используя процедуры MPI из библиотеки mpi.h и действуя по заданному человеком алгоритму, определяет, к какому из параллельных потоков она относится.
4. Программист «закладывает» в одну mpi программу все ветви, соответствующие различным параллельным потокам. При этом каждая ветвь жестко привязана к своему потоку (механизм MPI может быть включен и выключен только один раз в каждой копии параллельной программы).
5. Параллельно выполняемые ветви mpi программы могут обмениваться данными с помощью процедур приёма – передачи сообщений из библиотеки mpi.h.
6. Для согласования (синхронизации) процесса решения общей задачи различными ветвями mpi программы программист может использовать соответствующие процедуры из библиотеки mpi.h.

Представленные характеристики позволяют сделать вывод, что MPI технология однозначно используется человеком. Программистом при разработке параллельной программы. Пользователем при запуске mpi программы.

В ходе данного занятия мы познакомимся с альтернативным подходом к разработке параллельных программ. Альтернативная технология известна как OpenMP. Сразу отметим главное. Перед программистом не стоит выбор: что использовать MPI или OpenMP. Дело в том, что **инструменты OpenMP могут использоваться внутри любой программы, как обычной линейной, так и параллельной программы, созданной по технологии MPI.**

Главные отличительные особенности технологии OpenMP:

1. Задача создания параллельного кода возлагается на компилятор. Программист должен только определить участок программы, который надо разбить на несколько параллельных ветвей, и указать параметры распараллеливания (например, количество ветвей).
2. Параллельные ветви могут работать с одними и теми же данными.
3. В программу можно включать несколько участков, разбиваемых на параллельные ветви.
4. При переходе с одного суперкомпьютера на другой программу необходимо заново компилировать и собирать (конкретный компилятор настроен на конкретную операционную среду).
5. Для решения задач синхронизации, обмена данными и т.д. могут быть использованы процедуры из библиотеки omp.h.

Таким образом, **технология OpenMP даёт программисту дополнительные возможности для создания параллельных программ.**

OpenMP может использоваться совместно с другими технологиями параллельного программирования, например, с MPI. Обычно в этом случае MPI используется для распределения работы между несколькими вычислительными узлами, а OpenMP затем используется для распараллеливания на одном узле.

1. Модель параллельной программы OpenMP

При выполнении параллельной программы работа начинается с инициализации и выполнения главного потока, который по мере необходимости создает и выполняет параллельные потоки, передавая им необходимые данные. При разработке параллельной программы желательно выделять такие области распараллеливания, в которых можно организовать выполнение независимых параллельных потоков. Для обмена данными между параллельными потоками в OpenMP используются общие переменные. При обращении к общим переменным в различных параллельных потоках возможно возникновение конфликтных ситуаций при доступе к данным. Для предотвращения конфликтов можно воспользоваться процедурой синхронизации.

Выполнение параллельных потоков в параллельной области программы начинается с их инициализации. Она заключается в создании дескрипторов порождаемых потоков и

копировании всех данных из области данных главного потока в области данных создаваемых параллельных потоков. После порождения потоки нумеруются последовательными натуральными числами, причем главный поток имеет номер 0.

После завершения выполнения параллельных потоков управление программой вновь передается главному потоку. При этом возникает проблема корректной передачи данных от параллельных потоков главному. Здесь важную роль играет синхронизация завершения работы параллельных потоков, поскольку в силу целого ряда обстоятельств время выполнения даже одинаковых по трудоемкости параллельных потоков непредсказуемо (оно определяется как историей конкуренции параллельных процессов, так и текущим состоянием вычислительной системы). При выполнении операции синхронизации параллельные потоки, уже завершившие свое выполнение, простаивают и ожидают завершения работы самого последнего потока. Естественно, при этом неизбежна потеря эффективности работы параллельной программы.

Для того чтобы получить параллельную версию, сначала необходимо определить ресурс параллелизма программы, то есть, найти в ней участки, которые могут выполняться независимо разными нитями.

Как показывает практика, наибольший ресурс параллелизма в программах сосредоточен в циклах. Поэтому наиболее распространенным способом распараллеливания является то или иное распределение итераций циклов. Если между итерациями некоторого цикла нет информационных зависимостей, то их можно каким-либо способом раздать разным процессорам для одновременного исполнения.

Различные способы распределения итераций позволяют добиваться максимально равномерной загрузки нитей, между которыми распределяются итерации цикла.

Статический способ распределения итераций позволяет уже в момент написания программы точно определить, какой нити достанутся какие итерации. Однако он не учитывает текущей загруженности процессоров, соотношения времён выполнения различных итераций и некоторых других факторов. Эти факторы в той или иной степени учитываются динамическими способами распределения итераций. Кроме того, возможно отложить решение по способу распределения итераций на время выполнения программы (например, выбирать его, исходя из текущей загруженности нитей) или возложить выбор распределения на компилятор и/или систему выполнения.

Обмен данными в OpenMP происходит через общие переменные. Это приводит к необходимости разграничения одновременного доступа разных нитей к общим данным. Для этого предусмотрены достаточно развитые средства синхронизации.

2. Директивы и функции OpenMP

Значительная часть функциональности OpenMP реализуется при помощи директив компилятору. Они должны быть явно вставлены пользователем, что позволит выполнять программу в параллельном режиме. Директивы OpenMP в программах на языке C/C++ и являются указаниями препроцессору, начинающимися с ***#pragma omp***. Формат директивы на C/C++:

#pragma omp <имя директивы> [опция [[,] опция ...]

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

После некоторых опций может следовать список переменных, разделяемых запятыми. Каждая директива может иметь несколько дополнительных атрибутов – опций (clause). Отдельно специфицируются опции для назначения классов переменных, которые могут быть атрибутами различных директив.

Опция (clause) – это необязательный модификатор директивы, влияющий на ее поведение. Списки опций, поддерживаемые каждой директивой, различаются, а пять директив (master, critical, flush, ordered и atomic) вообще не поддерживают опции.

OpenMP поддерживает директивы parallel, for, parallelfor, section, sections, single, master, critical, flush, ordered, atomic и ряд других, которые определяют механизмы разделения работы или конструкции синхронизации.

Все директивы OpenMP можно разделить на 3 категории: определение параллельной области, распределение работы, синхронизация.

Чтобы задействовать функции библиотеки OpenMP периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл omp.h. Если вы используете в приложении только OpenMP-директивы, включать этот файл не требуется. Функции назначения параметров имеют приоритет над соответствующими переменными окружения.

Все функции, используемые в OpenMP, начинаются с префикса (приставки) omp_. Если пользователь не будет использовать в программе имён, начинающихся с такого префикса, то конфликтов с объектами OpenMP заведомо не будет. В языке C/C++, кроме того, является существенным регистр символов в названиях функций. Названия функций OpenMP записываются строчными буквами.

3. Простейшая OpenMP программа

3.1. Описание программы

Параллельная область задается при помощи директивы `#pragma omp parallel`. Разделяемое действие заключено в операторные скобки { }. Количество нитей задается с помощью процедуры `omp_set_num_threads()`, которая определена в библиотеке `omp.h`.

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

using namespace std;
int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        cout << "Hello world! ";
    }
    cout << endl;
}
```

В данном примере создается 4 нити, каждая из которых выводит на экран строку «Hello world!».

Желтым цветом выделены используемые в данном случае элементы OpenMP.

Следует отметить, что поскольку директива не является командой, после неё не ставится «;».

3.2. Компиляция и выполнение OpenMP программы

На рисунке показано, как происходит компиляция и выполнение OpenMP программы в среде суперкомпьютера.

```
bkiselev@master.unicluster ~/lesson11 $ mpic++ omp-hello.cpp -fopenmp -o omp-hello
bkiselev@master.unicluster ~/lesson11 $
bkiselev@master.unicluster ~/lesson11 $ mpirun -mca btl_openib_warn_no_device_params_found 0 -np 1 omp-hello
Hello world! Hello world! Hello world! Hello world!
bkiselev@master.unicluster ~/lesson11 $ |
```

При компиляции программы, использующей OpenMP, в командной строке обращения должен быть указан ключ **-fopenmp**. В остальном нет отличий по сравнению с «чистой» MPI программой.

Запуск программы, использующей OpenMP, не отличается от запуска программы MPI. Следует обратить внимание, что в приведенном примере количество иницилируемых при запуске потоков (ключ **-np**) равно 1. Тем не менее, описанное действие повторяется 4 раза.

4. Параллельные секции, переменные среды и замер времени

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

Рассмотрим возможные опции данной директивы.

num_threads (целочисленное выражение) – задание количества нитей, которые будут выполнять параллельную область; по умолчанию выбирается последнее значение, установленное с помощью функции *omp_set_num_threads()*, или значение переменной *OMP_NUM_THREADS*.

if(условие) – определяет условие выполнения параллельных потоков в последующем параллельном структурном блоке; если условие принимает значение истина, то потоки в последующем параллельном структурном блоке выполняются, в противном случае не выполняются.

shared(список) – задает список переменных, размещающихся в одной и той же области памяти для всех потоков.

private(список) – задает список переменных, локальных для каждого из параллельных потоков; в каждом из потоков эти переменные имеют собственные значения и относятся к различным областям памяти:

Локальным областям памяти каждого конкретного параллельного потока.

default(shared | none) – всем переменным параллельной области, которым явно не назначен класс *shared*; *none* означает, что всем переменным параллельной области должен быть явно назначен класс.

firstprivate(список) – задает список переменных, для которых порождается локальная копия для каждой нити, значения этих переменных инициализируются их значениями в нити-мастере.

copyin(список) – определяет список локальных переменных, которым присваиваются значения из одноименных общих переменных, заданных в глобальном потоке.

reduction(оператор: список) – определяется оператор - операции (+, -, *, / и т. п.) или функции, для которых будут вычисляться соответствующие частичные значения в параллельных потоках последующего параллельного структурного блока; кроме того, определяется список локальных переменных, в котором будут сохраняться соответствующие частичные значения; после завершения всех параллельных процессов частичные значения складываются (вычитаются, перемножаются и т. п.), и результат сохраняется в одноименной общей переменной.

Пример

```
#include <stdio.h>

int main()
{
    int n = 0;
    #pragma omp parallel num_threads (128) reduction (+: n)
    {
        n++;
        printf("Текущее значение n: %d\n", n);
    }
    printf("Текущее значение n: %d\n", n);
    return 0;
}
```

В данном примере каждая нить инициализирует переменную `n` значением 0. Затем все нити увеличивают это значение на 1. При выходе из параллельной области значение `n` будет равно количеству порожденных нитей.

В OpenMP предусмотрены функции для работы с системным таймером. Функция **`omp_get_wtime()`** возвращает в вызвавшей нити астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка.

Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменён за время существования процесса. Таймеры разных нитей могут быть не синхронизированы и выдавать различные значения.

Функция **`omp_get_wtick()`** возвращает в вызвавшей нити разрешение таймера в секундах. Это время можно рассматривать как меру точности таймера.

```
#include <stdio.h>
#include <omp.h>
int main()
{
    double time-start, time-end;
    time-start= omp_get_wtime();
    time-end= omp_get_wtime();
    printf("Время выполнения %lf\n", end_time-start_time);
    return 0;
}
```

В примере (язык C) происходит замер начального времени, а затем конечного. Результатом программы является время на замер времени, которое дает разность времен.

В параллельной области каждой имеющейся нитью может быть порождена параллельная секция и последующее их соединение с сохранением главенства порождающей нити. Число нитей в параллельной секции можно задавать с помощью функции **`omp_set_num_threads()`**. Эта функция устанавливает значение переменной **`OMP_NUM_THREADS`**.

В некоторых случаях система может динамически изменять количество нитей, используемых для выполнения параллельной области, например, для оптимизации использования ресурсов системы. Это разрешено делать, если значение переменной среды **`OMP_DYNAMIC`** установлено в 1, что можно сделать с помощью функции **`omp_set_dynamic()`**. Получить значение переменной **`OMP_DYNAMIC`** можно с помощью функции **`omp_get_dynamic()`**.

Стратегию обработки вложенных секций можно менять с помощью задания значений переменной среды **`OMP_NESTED`** функцией **`omp_set_nested()`**, где в качестве параметра задается 0 или 1. Данная функция разрешает или запрещает вложенный параллелизм. Если вложенный параллелизм разрешён, то каждая нить, в которой встретится описание параллельной области, породит для её выполнения новую группу нитей. Сама породившая нить станет в новой группе нитью-мастером. Если система не поддерживает вложенный параллелизм, данная функция не будет иметь эффекта. Получить значение переменной **`OMP_NESTED`** можно с помощью функции **`omp_get_nested()`**.

Функция **`omp_get_max_threads()`** возвращает максимально допустимое число нитей для использования в следующей параллельной области.

Функция **`omp_get_num_procs()`** возвращает количество процессоров, доступных для использования программе пользователя на момент вызова. Нужно учитывать, что количество доступных процессоров может динамически изменяться.

Функция **`omp_in_parallel()`** возвращает 1, если она была вызвана из активной параллельной области программы.

Переменная ***OMP_MAX_ACTIVE_LEVELS*** задаёт максимально допустимое количество вложенных параллельных областей. Значением может быть установлено при помощи вызова функции ***omp_set_max_active_levels()***.

Если значение *max* превышает максимально допустимое в системе, будет установлено максимально допустимое в системе значение. При вызове из параллельной области результат выполнения зависит от реализации.

Значение переменной ***OMP_MAX_ACTIVE_LEVELS*** может быть получено при помощи вызова функции ***omp_get_max_active_levels()***.

Функция ***omp_get_level()*** выдаёт для вызвавшей нити количество вложенных параллельных областей в данном месте кода. При вызове из последовательной области функция возвращает значение 0.

Функция ***omp_get_ancestor_thread_num()*** возвращает для уровня вложенности параллельных областей, заданного параметром *level*, номер нити, породившей данную нить. Если *level* меньше нуля или больше текущего уровня вложенности, возвращается -1. Если *level*=0, функция вернёт 0, а если *level*=***omp_get_level()***, вызов эквивалентен вызову функции ***omp_get_thread_num()***.

Функция ***omp_get_team_size()*** возвращает для заданного параметром *level* уровня вложенности параллельных областей количество нитей, порождённых одной родительской нитью. Если *level* меньше нуля или больше текущего уровня вложенности, возвращается -1. Если *level*=0, функция вернёт 1, а если *level*=***omp_get_level()***, вызов эквивалентен вызову функции ***omp_get_num_threads()***.

Функция ***omp_get_active_level()*** возвращает для вызвавшей нити количество вложенных параллельных областей, обрабатываемых более чем одной нитью, в данном месте кода. При вызове из последовательной области возвращает значение 0.

Переменная среды ***OMP_STACKSIZE*** задаёт размер стека для создаваемых из программы нитей. Значение переменной может задаваться в виде *size* / *sizeB* / *sizeK* / *sizeM* / *sizeG*, где *size* – положительное целое число, а буквы B, K, M, G задают соответственно, байты, килобайты, мегабайты и гигабайты. Если ни одной из этих букв не указано, размер задаётся в килобайтах. Если задан неправильный формат или невозможно выделить запрошенный размер стека, результат будет зависеть от реализации.

Переменная среды ***OMP_WAIT_POLICY*** задаёт поведение ждущих процессов. Если задано значение ***ACTIVE***, то ждущему процессу будут выделяться циклы процессорного времени, а при значении ***PASSIVE*** ждущий процесс может быть отправлен в спящий режим, при этом процессор может быть назначен другим процессам.

Переменная среды ***OMP_THREAD_LIMIT*** задаёт максимальное число нитей, допустимых в программе. Если значение переменной не является положительным целым числом или превышает максимально допустимое в системе число процессов, поведение программы будет зависеть от реализации. Значение переменной может быть получено при помощи процедуры ***omp_get_thread_limit()***

5. Варианты распределения работы между нитями параллельной программы

5.1. Низкоуровневое программирование

Все нити в параллельной области нумеруются последовательными целыми числами от 0 до N-1, где N — количество нитей, выполняющих данную область. Аналогично нумеруются параллельные процессы в MPI программе.

Можно программировать на самом низком уровне, распределяя работу с помощью функций ***omp_get_thread_num()*** и ***omp_get_num_threads()***, возвращающих номер нити и общее количество порождённых нитей в текущей параллельной области, соответственно.

Вызов функции ***omp_get_thread_num()*** позволяет нити получить свой уникальный номер в текущей параллельной области. В MPI программе поток получает аналогичную информацию с помощью процедуры ***MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank)***.

Вызов функции `omp_get_num_threads()` позволяет нити получить количество нитей в текущей параллельной области. В MPI есть аналогичная функция ***MPI_Comm_size(MPI_COMM_WORLD, &ProcNum)***.

Использование функций ***omp_get_thread_num()*** и ***omp_get_num_threads()*** позволяет назначать каждой нити свой кусок кода для выполнения. Однако использование этого стиля программирования в OpenMP далеко не всегда оправдано – разработчик в этом случае должен явно организовывать синхронизацию доступа к общим данным.

Пример.

```
#include <stdio.h>
#include <omp.h>
int main()
{
    60
    int count, num;
    #pragma omp parallel
    {
        count=omp_get_num_threads();
        num=omp_get_thread_num();
        if (num == 0) printf("Всего нитей: %d\n", count);
        else printf("Нить номер %d\n", num);
    }
    return 0;
}
```

В данном примере в параллельной области определен условный оператор, в результате выполнения которого главный поток выведет на экран количество нитей, а остальные потоки свой номер.

В MPI программах мы использовали аналогичный прием. Например ,

```
if(ProcRank==1+i*j*Nrow+k*Nrow*Nrow)
    { ... }
```

при распараллеливании процесса построения 3-d модели кристаллической решетки.

5.2.Распараллеливание оператора цикла

Циклы, как говорилось раньше, являются важнейшим ресурсом для распараллеливания процесса обработки информации.

Если в параллельной области встретился оператор цикла без дополнительных указаний, то он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла. Для распределения операций тела цикла между различными потоками при программировании по технологии MPI мы каждому потоку определяли конкретный диапазон изменения параметра цикла. Например,

```
for(i=rank+1; i <= n; i+=size)
{ x = h*(i-0.5); sum = sum + f(x); }
```

Для распределения операций тела цикла между различными нитями в OpenMP можно использовать **директиву *for***. Эта директива относится к следующему за ней блоку, содержащему **оператор *for***.

На языке Си директива выглядит следующим образом:

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

Рассмотрим опции данной директивы.

private(список) – задает 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 (с номером 0).

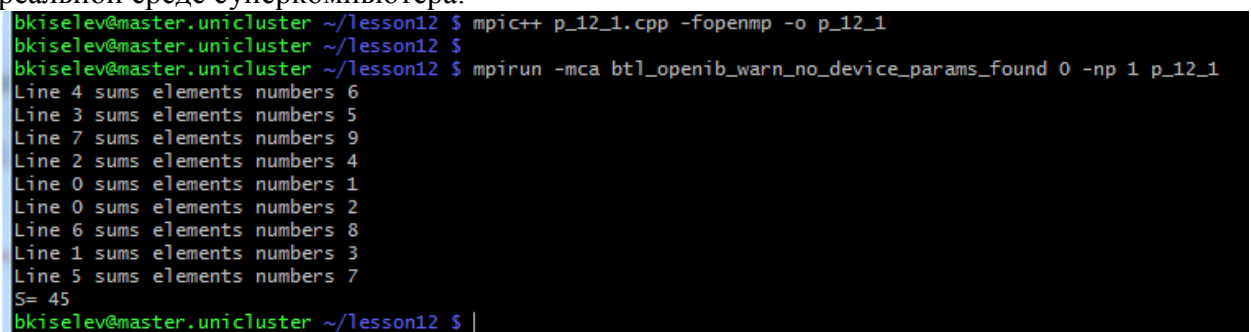
lastprivate(cnucok) – переменным, перечисленным в списке, присваивается результат с последнего витка цикла.

reduction(oneпамор:cnucok) – определяется оператор - операции (+, -, *, / и т. п.) или функции, для которых будут вычисляться соответствующие частичные значения в параллельных потоках последующего параллельного структурного блока; кроме того, определяется список локальных переменных, в котором будут сохраняться соответствующие частичные значения. После завершения всех параллельных процессов частичные значения складываются (вычитаются, перемножаются и т. п.), и результат сохраняется в одноименной общей переменной.

Пример.

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int s, i, n;
    s=0;
    #pragma omp parallel private (i,n) reduction(+:s)
    {
        n=omp_get_thread_num();
        #pragma omp for
        for (i=1; i<10; i++)
        {
            s=s+i;
            printf("Нить %d сложила элементы с номером %d\n", n, i);
        }
        printf("S= %d\n", s);
        return 0;
    }
}
```

Пример демонстрирует нахождение суммы чисел от 1 до 10 с помощью распараллеливания цикла. На экран выводится номер нити и номера итераций цикла, которые эта нить выполнила. На рисунке видно, как выполняется эта программа в реальной среде суперкомпьютера.



```
bkiselev@master.unicluster ~/lesson12 $ mpic++ p_12_1.cpp -fopenmp -o p_12_1
bkiselev@master.unicluster ~/lesson12 $
bkiselev@master.unicluster ~/lesson12 $ mpirun -mca btl_openib_warn_no_device_params_found 0 -np 1 p_12_1
Line 4 sums elements numbers 6
Line 3 sums elements numbers 5
Line 7 sums elements numbers 9
Line 2 sums elements numbers 4
Line 0 sums elements numbers 1
Line 0 sums elements numbers 2
Line 6 sums elements numbers 8
Line 1 sums elements numbers 3
Line 5 sums elements numbers 7
S= 45
bkiselev@master.unicluster ~/lesson12 $ |
```

schedule(type[, chunk]) – опция задаёт, каким образом операции тела цикла распределяются между нитями.

collapse(n) — опция указывает, что ***n*** последовательных вложенных циклов ассоциируется с данной директивой. Для циклов образуется общее пространство операций, 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 операции 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.

Предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит.

Нельзя использовать побочный выход из параллельного цикла.

Если директива параллельного выполнения стоит перед набором вложенных циклов, завершающихся одним оператором, то директива действует только на самый внешний цикл.

Параметрами опции *schedule* являются следующие: *static* – распределение операций цикла; размер блока – *chunk*.

Первый блок из *chunk* операций выполняет нулевая нить, второй блок – следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение *chunk* не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции операций распределяются между нитями.

dynamic – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает *chunk* операций (по умолчанию *chunk*=1), та нить, которая заканчивает выполнение своей порции операций, получает первую свободную порцию из *chunk* операций. Освободившиеся нити получают новые порции операций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше операций тела цикла, чем все остальные.

guided – динамическое распределение операций, при котором размер порции уменьшается с некоторого начального значения до величины *chunk* (по умолчанию *chunk*=1) пропорционально количеству ещё не распределённых операций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество операций в последней порции может оказаться меньше значения *chunk*.

auto – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр *chunk* при этом не задаётся.

runtime – способ распределения итераций выбирается во время работы программы по значению переменной среды **OMP_SCHEDULE**. Параметр *chunk* при этом не задаётся.

При распараллеливании цикла с помощью директив OpenMP следует убедиться в том, что его операции тела цикла не имеют зависимостей, и их можно выполнять в любом порядке. Несоблюдение данного требования приведет к получению некорректного результата.

Например, недопустимо, чтобы выполнение тела цикла при одном значении параметра зависело от результата выполнения тела цикла при другом значении параметра.

6. Распараллеливание линейной программы директивами OpenMP

Параллельную программу по технологии MPI, создаваемую на основе существующей линейной программы, необходимо заново перепрограммировать. Это связано с тем, что в каждом из параллельных потоков механизм MPI может быть включен только один раз.

При использовании технологии OpenMP оказывается возможным поэтапное распараллеливание отдельных участков линейной программы, допускающих применение соответствующих директив.

В качестве применения последнего подхода рассмотрим процедуру построения 3-d модели поверхности эллипсоида. Эта процедура была нами разработана ранее.

Особенностью этой процедуры было то, что сначала рассчитывалась верхняя часть поверхности эллипсоида (для $z > 0$). При этом в расчетах были задействованы на каждом

шаге построения 4 точки поверхности, характеристики которых рассчитывались по рекуррентным формулам. В этом случае применение директив распараллеливания либо вообще невозможно, либо потребует кардинальной переработки алгоритма.

На втором этапе строится нижняя часть ($z < 0$). При рассмотрении этой части процедуры мы можем выделить, например, такую последовательность операторов:

```
//Добавление южного полушария
NN= node;
for (i=NN1; i<NN; i++){
    X[node]=X[i];
    Y[node]=Y[i];
    Z[node]=2*z0-Z[i];
    node++;
}
```

В этом цикле происходит однозначное отображение точек верхней части поверхности на нижнюю. Т.е., в принципе допустимо распараллеливание. «Мешает» счетчик node. Чтобы убрать это препятствие перепишем этот участок по-другому.

```
NN= node;
for (i=NN1; i<NN; i++){
    X[NN+(i-NN1)]=X[i];
    Y[NN+(i-NN1)]=Y[i];
    Z[NN+(i-NN1)]=2*z0-Z[i];
}
node=NN+(NN-NN1);
```

Проверка показывает, что внесенные изменения не влияют на выполнение программы

Но такой цикл может быть разделен на несколько нитей.

```
NN= node;
#pragma omp parallel for shared(X,Y,Z) private(i)
for (i=NN1; i<NN; i++){
    X[NN+(i-NN1)]=X[i];
    Y[NN+(i-NN1)]=Y[i];
    Z[NN+(i-NN1)]=2*z0-Z[i];
}
node=NN+(NN-NN1);
```

Проверка на суперкомпьютере показывает что этот код работает.

Следующим претендентом является цикл отображения триангуляционной сети

```
NK= k;
for (i=NK1; i<NK; i++){
    Itr[k]=Itr[i]+NN-NN1;
    Jtr[k]=Jtr[i]+NN-NN1;
    Ktr[k]=Ktr[i]+NN-NN1;
    k++;
}
```

Вносим аналогичные коррективы:

```
NK= k;
#pragma omp parallel for shared(Itr,Jtr,Ktr) private(i)
for (i=NK1; i<NK; i++){
    Itr[NK+(i-NK1)]=Itr[i]+NN-NN1;
    Jtr[NK+(i-NK1)]=Jtr[i]+NN-NN1;
    Ktr[NK+(i-NK1)]=Ktr[i]+NN-NN1;
}
k=NK+(NK-NK1);
```

7. Примеры программ с использованием технологии OpenMP

7.1. Параллельное перемножение двух квадратных матриц

Первая программа реализует операцию перемножения двух квадратных матриц. В последовательной области происходит инициализация массивов и замер времени. Затем порождается параллельная область, исходные и результирующий массивы являются общими переменными, а параметры циклов частными. После выхода из параллельной секции снова происходит замер времени, вывод результата и общего времени выполнения программы на экран.

Программа:

```
#include <stdio.h>
#include <omp.h>
#define N 1000
int main()
{
    double A[N][N], B[N][N], C[N][N];
    int i, j, k;
    double start, end;
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
        {
            A[i][j]=i+j;
            B[i][j]=i*j;
        }
    start=omp_get_wtime();
    #pragma omp parallel for shared(A,B,C) private(i, j, k)
    for (i=0; i<N; i++){
        for (j=0; j<N; j++) {
            C[i][j]=0;
            for (k=0; k<N; k++)
                C[i][j]+=A[i][k]*B[k][j];
        }
    }
    end=omp_get_wtime();
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            printf("C[%d][%d]=%lf\n", i, j, C[i][j]);
    printf("Time %lf\n", end-start);
    return 0;
}
```

7.2. Приближенное вычисление определенного интеграла

В следующем примере вычисляется определенный интеграл от функции $\sin(x)$ методом трапеций. В параллельной области вычисляется значение функции для каждого интервала разбиения и разграничивается доступ к операции сложения директивой **atomic**.

Программа:

```
#include <cstdlib>
#include <stdio.h>
#include <math.h>
double a,b,h;
double f(int j)
{
    return sin(a+j*h);
}
```

```

int main()
{
double sum,result;
int i,N;
sum=0;
result=0;
printf("Vvedite a,b,N\n");
scanf("%lf %lf %d", a ,b ,N);
h=(b-a)/N;
#pragma omp parallel for private (i) shared (N)
for (i=1; i<N; i++)
{
#pragma omp atomic
sum+=f(i);
}
result=h*((f(0)+f(N))/2+sum);
printf("Znachenie %0.5lf \n",result);
return 0;
}

```

7.3. Использование OpenMP в MPI программе

Рассмотрим пример совместного использования инструментов OpenMP и MPI в одной программе.

```

#include <stdio.h>
#include <mpi.h>

using namespace std;
int main(int argc, char* argv[])
{
    int rank, size, n, i;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    n=0;
    for(i=0; i<size; i++)
        if (rank==i){
            #pragma omp parallel num_threads (i+1) reduction (+: n)
            {
                n++;
                printf("n= %d\n", n);
            }
            printf("n= %d i= %d\n", n, i);
        }
    MPI_Finalize();
}

```

В этой программе зеленым цветом выделены элементы MPI, а желтым OpenMP. Программа запускается на нескольких потоках количество которых определяется командой запуска. В ходе работы каждая ветвь MPI программы в свою очередь разбивается на несколько нитей, количество которых равно номеру MPI ветви плюс 1.

На рисунке показано, как выполняется компиляция, запускается программа на 4-х потоках и показан результат. Результаты не упорядочены, т.к. работа MPI не синхронизирована.

```
bkiselev@master.unicluster ~/lesson11 $ mpic++ mpi+omp.cpp -fopenmp -o mpi+omp
bkiselev@master.unicluster ~/lesson11 $
bkiselev@master.unicluster ~/lesson11 $ mpirun -mca btl_openib_warn_no_device_params_found 0 -np 4 mpi+omp
n= 1 i= 0
n= 2 i= 1
n= 4 i= 3
n= 3 i= 2
bkiselev@master.unicluster ~/lesson11 $
```

ЗАДАНИЕ

Разработайте с помощью OpenMP параллельную программу приближенного вычисления числа π .