САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

«OpenMP»

Выполнил(а): Султанов Мирзомансурхон Махсудович

студ. гр. М313Д

Санкт-Петербург

2020

Цель работы: знакомство со стандартом распараллеливания команд OpenMP.
Инструментарий и требования к работе: рекомендуется использовать C, C++. Возможно использовать Python и Java.

Теоретическая часть

OpenMP помогает писать программы, которые не требуют сильных изменений, чтобы перевести программу в параллельный режим. В целом, OpenMP – это открытый стандарт для написания параллельных программ для систем с общей памятью. Официально поддерживаются C, C++ и Fortran. Несмотря на это, можно найти реализации и для некоторых других языков, к примеру, Java. OpenMP даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Существует множество разновидностей параллельных вычислительных систем – многоядерные/многопроцессорные компьютеры, кластеры, системы на видеокартах и др. Библиотека OpenMP подходит только для программирования систем с общей памятью, где при этом используется параллелизм потоков. Потоки же создаются в рамках единственного процесса и имеют собственную память. Кроме того, все потоки имеют доступ к памяти процесса (см. рисунок 1).

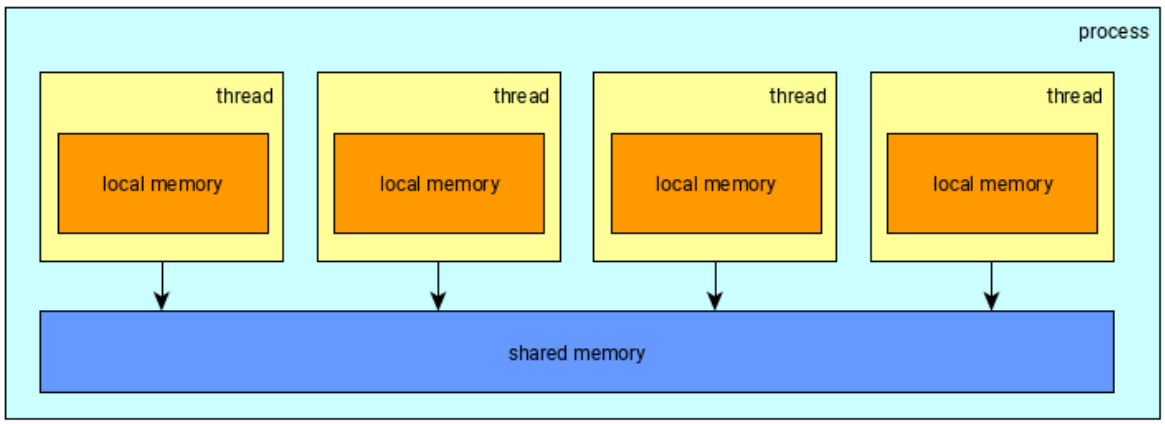

Рисунок 1 – модель памяти в OpenMP

В модели с разделяемой памятью взаимодействие потоков происходит через локальные переменные, доступ к которым имеет лишь соответствующий поток. При неправильном обращении с такими переменными в программе могут возникнуть ошибки соревнования (race condition). Ошибка возникает вследствие того, что потоки выполняются параллельно и соответственно последовательность доступа к разделяемым переменным может быть различна от одного запуска программы к другому. Для контроля ошибок соревнования работу потоков необходимо синхронизировать. Для разных же программ нужно по-разному синхронизировать переменные. Отчасти именно поэтому параллельное программирование — это бремя программистов, и оно не имеет аппаратного решения. Для решения этой проблемы используются такие примитивы синхронизации как критические секции, барьеры, атомарные операции и блокировки.

В варианте 4 я использовал директивы atomic и reduction, для того чтобы избежать состояние гонки.

Идея использования atomic заключается в следующем. Для каждого потока у нас будет свой локальный счётчик. А глобальный счётчик будет помечен как atomic, т.е. увеличивать этот счётчик может лишь один поток в одно время. Для остальных же потоков доступ к глобальному счётчику закрыт, и они будут ждать, когда доступ появится.

Принцип работы директивы reduction выглядит так. Сначала для каждой переменной создаются локальные копии в каждом потоке. Локальные копии инициализируются соответственно типу оператора. Для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или его аналоги. Все исходные значения переменных можно увидеть на рисунке 2. И наконец, над локальными копиями переменных после выполнения операторов параллельной области выполняется заданный оператор. Порядок выполнения не определён.

Оператор	Исходное значение переменной
+	0
*	1
-	0
&	~0 (каждый бит установлен)
|	0
^	0
&&	1
||	0

Рисунок 2 – Операторы reduction и их исходные значения переменных

Количество задаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне при помощи переменных окружения.

Для того, чтобы программа, написанная на C++, могла использовать библиотеки OpenMP, необходимо подключить заголовочный файл “omp.h”, а также во время компиляции через командную строку добавить опцию сборки –fopenmp для компиляторов gcc и g++ или установить соответствующий флаг в настройках проекта (для Visual Studio или CLion). Для некоторых компиляторов иногда требуется докачать дополнительные библиотеки, чтобы опция –fopenmp работала корректно.

В целом, ключевыми элементами OpenMP являются: конструкции для создания потоков (директива parallel); конструкции распределения работы между потоками (директивы DO/for и section); конструкции для управления работой с данными (выражения shared и private для определения класса памяти переменных); конструкции для синхронизации потоков (директивы critical, atomic и barrier); процедуры библиотеки поддержки времени выполнения (например, omp_get_thread_num); переменные окружения (например, OMP_NUM_THREADS).

После запуска программы создаётся единственный процесс, который начинает выполняться, как и обычная последовательная программа. Встретив параллельную область, задаваемую #pragma omp parallel, процесс порождает ряд потоков (их число можно задать явно, однако по умолчанию будет создано столько потоков, сколько в имеющейся системе вычислительных ядер). Для того чтобы указать количество потоков, нужно использовать функцию из библиотеки <omp.h>: omp_set_num_threads(num), где num – количество потоков. Границы параллельной области выделяются фигурными скобками, в конце области потоки уничтожаются. Схематично этот процесс изображён на 3 рисунке.

```
#include "omp.h"

int main() {
  // A - single thread
#pragma omp parallel
  {
    // B - many threads
  }
  // C - single thread
#pragma omp parallel
  {
    // D - many threads
  }
  // E - single thread
}
```

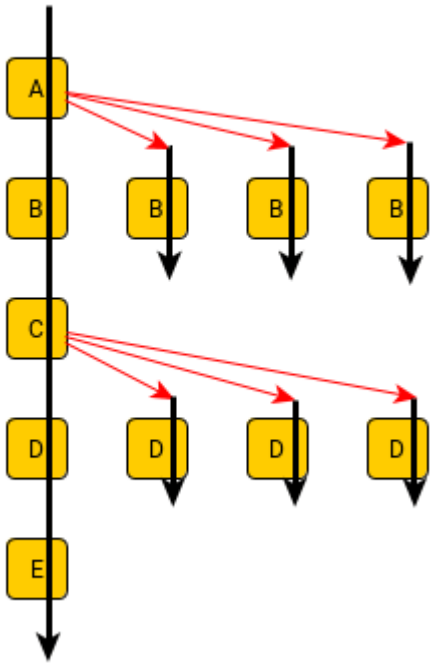

Рисунок 3 – директива omp parallel

Чёрными линии на рисунке – это время жизни потоков, а красные – этот момент порождения. Видно, что все потоки создаются одним(главным) потоком, который существует всё время работы процесса. Такой поток в OpenMP называется master, все остальные потоки многократно создаются и уничтожаются. Стоит отметить, что директивы parallel могут быть

вложенными, при этом в зависимости от настроек могут создаваться вложенные потоки.

Директива for используется для явного распараллеливания следующего цикла for, при этом каждая нить начинается со своего индекса. Если не указывать директиву, то цикл будет пройден каждой нитью полностью от начала и до конца.

Параметр schedule для директив с циклом (для таких, как for) нужен для планирования распределения итераций цикла между потоками. Существует пять различных типов параметра schedule: static, dynamic, guided, auto, runtime.

Для static вся совокупность загружаемых процессов разбивается на равные порции размера chunk, и эти порции последовательно распределяются между процессорами потоками с первого до последнего и т.д.

schedule(static, chunk)

Если chunk отсутствует, то OpenMP разделяет итерации на фрагменты примерно равного размера и распределяет не более одного фрагмента на каждый поток. Примеры показаны на рисунке 4.

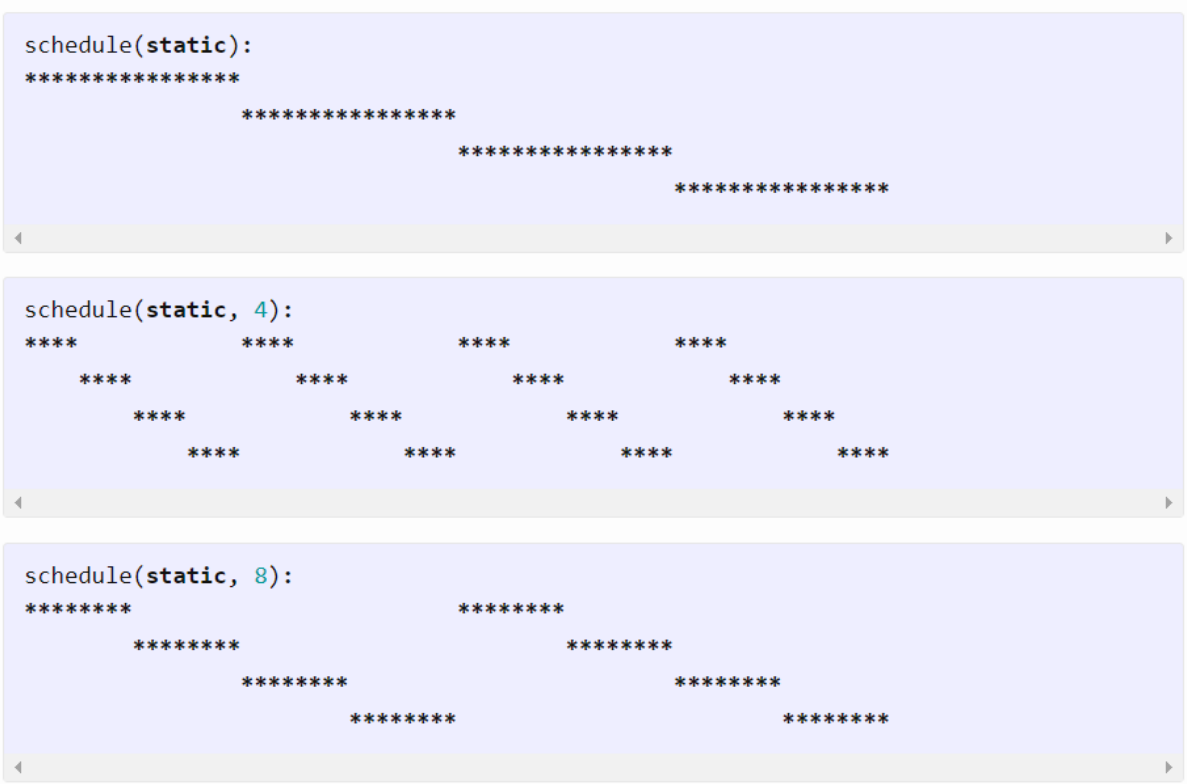

Рисунок 4 – Примеры работы static schedule

Для dynamic вся совокупность загружаемых процессов, как и в предыдущем варианте, разбивается на равные порции размера chunk, но эти порции загружаются последовательно в освободившиеся потоки (процессоры).

schedule(dynamic, chunk)

Если отсутствует chunk, вызывается schedule(dynamic, 1)

Примеры показаны на рисунке 5.

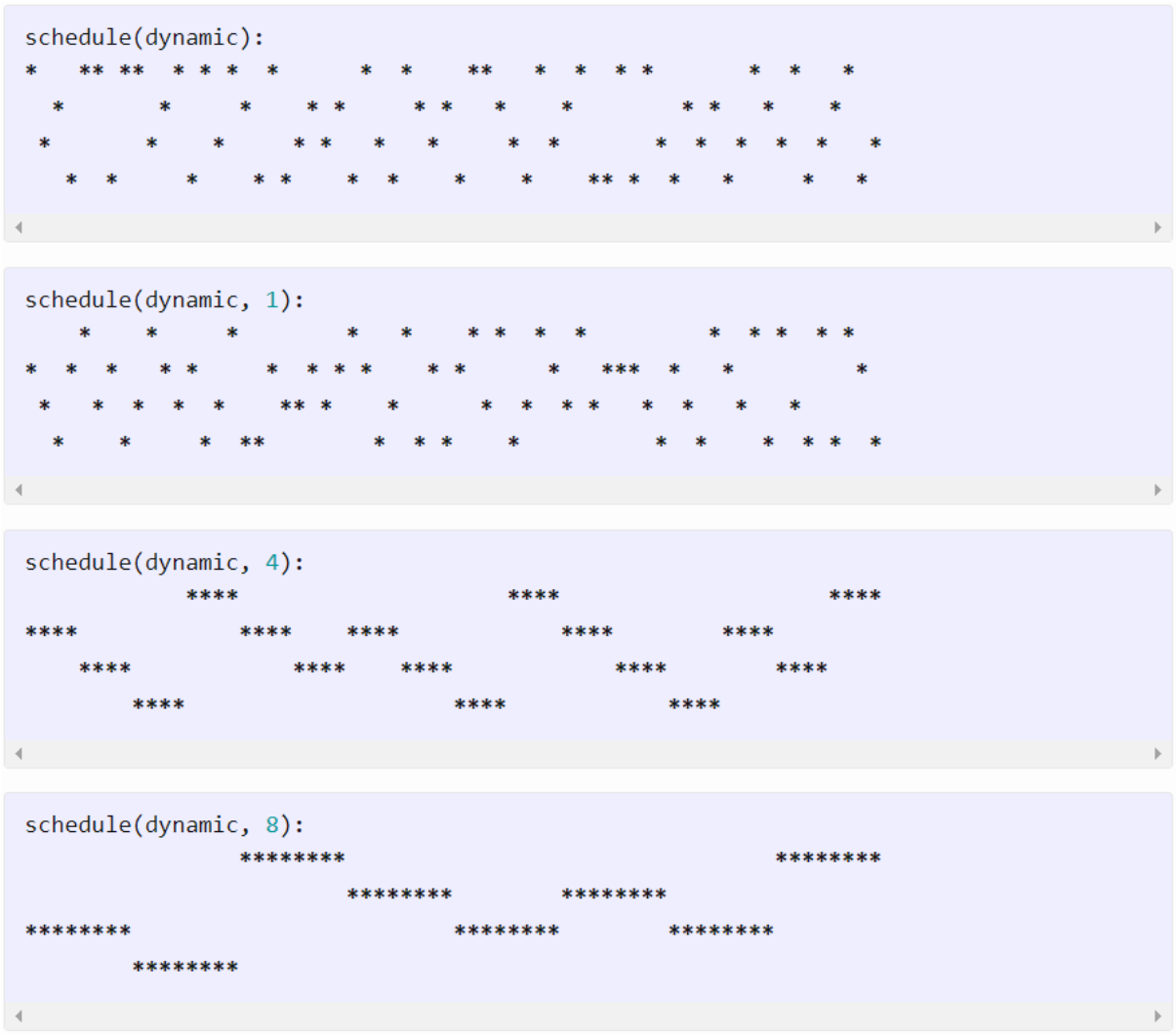

Рисунок 5 – Примеры работы dynamic schedule

Guided аналогичен dynamic. OpenMP снова делит итерации на части. Каждый поток выполняет часть итераций, а затем запрашивает другой фрагмент, пока не закончатся доступные фрагменты. Разница с типом динамического программирования заключается в размере блоков. Размер чанка пропорционален количеству неназначенных итераций, разделённому на количество потоков. Поэтому размер кусков уменьшается. Минимальный размер чанка устанавливает значение chunk, однако фрагмент, содержащий последнюю итерацию, может иметь размер меньше, чем chunk.

schedule(guided, chunk)

Если отсутствует chunk, вызывается schedule(guided, 1)

Примеры показаны на рисунке 6.

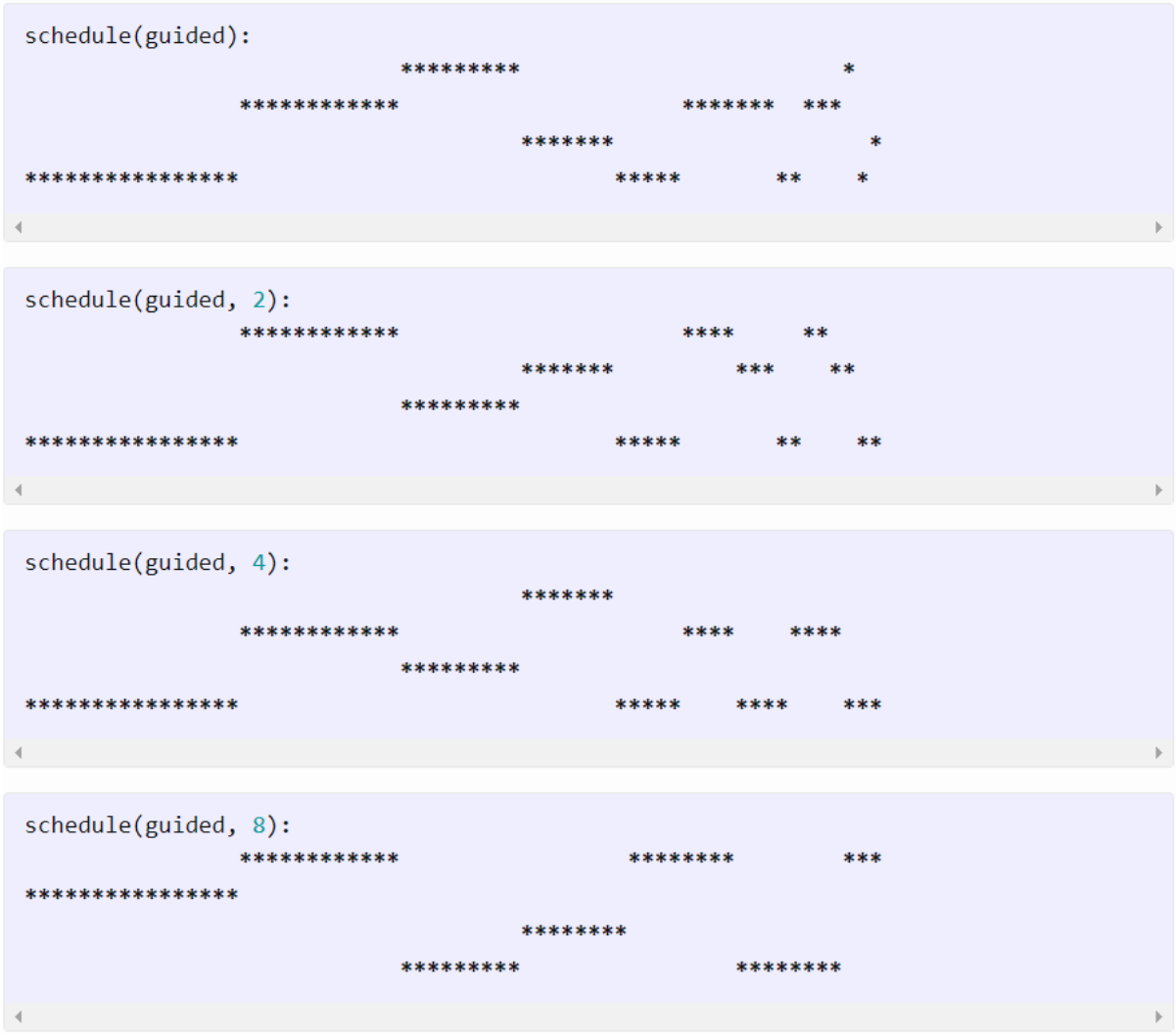

Рисунок 6 – примеры работы guided schedule

Auto – тип, при котором компилятор или исполняющая система сами выбирают один из трёх ранее рассказанных типов по своему усмотрению.

Runtime – этот тип выбирает один из трёх первых типов, который указан в переменной окружения OMP_SCHEDULE. Это полезно, если в зависимости от ситуации необходимо выбрать какой-то конкретный тип schedule.

Описание работы кода (4 вариант)

Подсчёт простых чисел в интервале от 2 до n осуществляется следующим образом. Перебираются все числа от 3 до n - 1 с шагом 2, так как чётные

числа очевидным образом являются составными. И для каждого числа проверяем делится ли оно нацело на число из интервала от 2 до $n^{0.5}$ так же с шагом 2. И если хотя бы раз поделилось, то тогда число составное. Если число простое, то увеличиваем счётчик на 1.

Сейчас я только описал последовательное решение программы. Чтобы получить параллельное решение достаточно распараллелить циклы таким образом, чтобы не было состояние гонки, то есть необходимо воспользоваться atomic или reduction. Проверим, как всё это влияет на время.

Графики времени работы программы

Вычисление времени будет проводиться на моём ноутбуке с процессором AMD Ryzen 3 3200U 2.60 GHz с 2 ядрами и 4 потоками. Поэтому будем рассматривать значения потоков от 1 до 4, так как только это мне доступно. Создадим таблицу времён в секундах с различным числов потоков и различными типами schedule. В каждой ячейке сделаем по три вычисления времени, дабы учесть погрешности. Ниже трёх вычислений будет представлено среднее арифметическое. Проверять всё будет на количестве простых чисел в интервале от 2 до 10000000 (10 миллионов).

Для того чтобы работал OpenMP были установлены необходимые флаги в CMakeLists.txt.

Таблица №1 – Вычисления времени в секундах для решения с atomic

Schedule\потоки	1 поток	2 поток	3 поток	4 поток
Static	6.42223	4.48839	3.8156	3.61507
	6.4054	4.43716	3.78945	3.45145
	6.29141	4.68102	4.01943	3.55204
	≈6.373	≈4.534	≈3.874	≈3.539
Dynamic	6.91185	4.41362	3.88512	3.62928
	6.93142	4.43452	3.84322	3.65238

	6.94791 ≈6.930	4.26473 ≈4.371	3.75148 ≈3.826	3.55116 ≈3.611
Guided	6.70914 6.70496 6.57236 ≈6.662	4.09439 4.10142 4.2245 ≈4.140	3.53049 3.76385 3.80964 ≈3700	3.42182 3.43182 3.49192 ≈3.448
Без OpenMP	6.44776 6.41573 6.6044 ≈6489			

Видно по таблице, что с увеличением потоков при одинаковом параметре schedule уменьшается время выполнения программы, что вполне логично. Но с каждым переходом выгода во времени становится всё меньше. Предполагаю, что это связано с тем, что очередь к атомарному элементу answer становится всё больше и из-за этого снижается эффективность во времени.

Легко заметить, что для данной задачи эффективнее всего подходит guided schedule, за ним идёт dynamic, и последний static.

В среднем, OpenMP с одним потоком требует немного больше времени нежели решение без OpenMP. Оно и понятно, помимо выполнения нашей задачи мы ещё тратим время на то, чтобы поддерживать один поток и из-за этого мы теряем во времени.

Листинг

tmain.cpp

```
#include <iostream>

#include <ctime>
```

```
#include <omp.h>


bool IsSimple(int n) {
    for (int j = 3; j * j <= n; j += 2)
        if (n % j == 0)
            return false;
    return true;
}
void WithOpenMP(int n, int num_threads) {
    double start_time = omp_get_wtime();
    omp_set_num_threads(num_threads);
    int answer = 0;


    #pragma omp parallel
    {
        int y = 0;
        #pragma omp for schedule(static)
        //#pragma omp for schedule(dynamic)
        //#pragma omp for schedule(guided)
        for (int num = 3; num < n; num += 2)
            if (IsSimple(num))
                y++;
```

```
        #pragma omp atomic
        answer += y;
    }
    /*
    #pragma omp parallel for schedule(static) reduction(+:answer)
    for (int num = 3; num < n; num += 2)
        if (IsSimple(num))
            answer++;
    */
    std::cout<<"Time:"<<omp_get_wtime()  - start_time<<" s"<<"\n";
    std::cout<<"Result:"<<answer<<"\n";
}
main() {
    int n = 10000000;
    /*
    std::cout<<"n = ";
    std::cin>>n;
    */
    for (int j = 0; j < 3; j++) {
        for (int i = 1; i <= 4; i++) {
            std::cout << "With " << i << " threads" << "\n";
            WithOpenMP(n, i);
```

```
            }

        }

    }
```