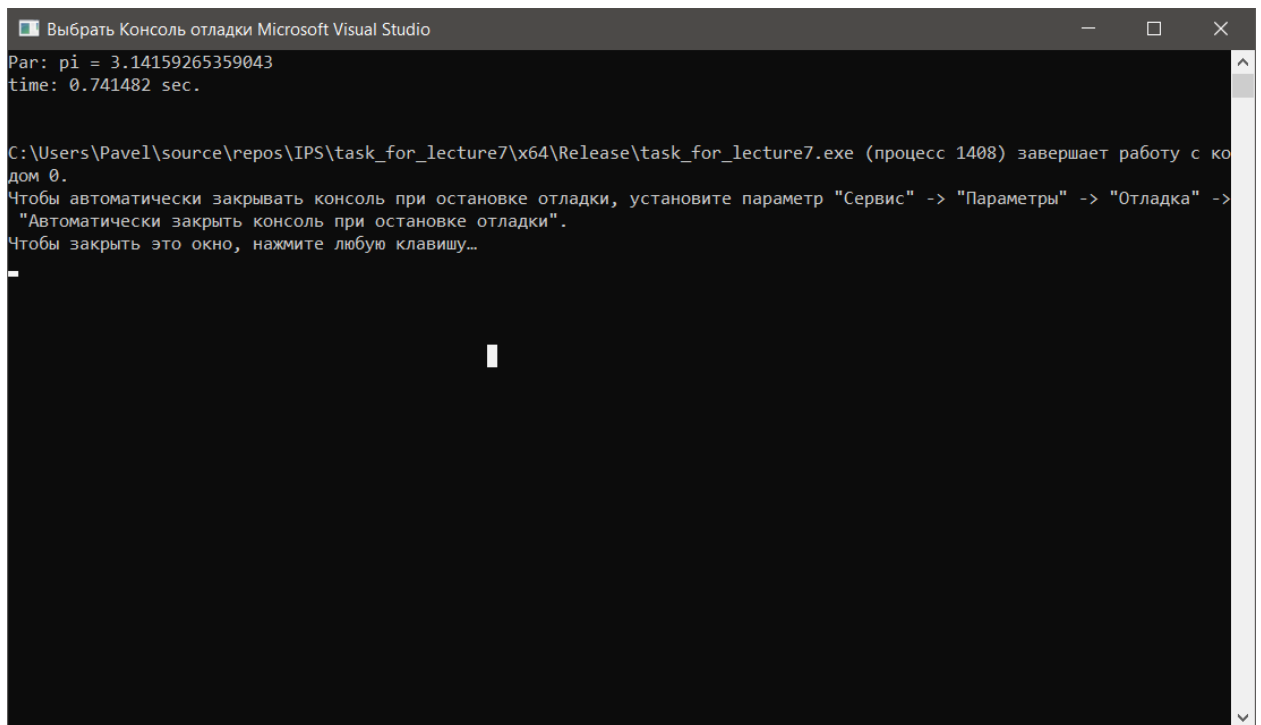


Программное обеспечение, использованное при выполнении: Visual Studio 2017, IPS 2019, Windows 10 – 64x

Процессор – четырехъядерный Intel Core i5 8250U с частотой 1.6 ГГц

Работа выполнялась на версии Release x64

1. Разберите последовательную программу по вычислению определенного интеграла [task_lecture_7.cpp](#). Введите в нее параллелизм с помощью OpenMP. Установите количество рабочих процессов равным 3, для этого используйте оператор **num_threads(num_of_threads)**. Не забудьте настроить в свойствах проекта поддержку стандарта OpenMP: **Свойства проекта** -> вкладка **C/C++** -> **Язык** -> **Поддержка OpenMP**.



```
Выбрать Консоль отладки Microsoft Visual Studio
Par: pi = 3.14159265359043
time: 0.741482 sec.

C:\Users\Pavel\source\repos\IPS\task_for_lecture7\x64\Release\task_for_lecture7.exe (процесс 1408) завершает работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, установите параметр "Сервис" -> "Параметры" -> "Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Чтобы закрыть это окно, нажмите любую клавишу...
```

Рисунок 1 – пример первого запуска программы

Введем в программу параллелизм с помощью директивы `#pragma omp parallel for` `num_threads(num_of_threads)`

```
#pragma omp parallel for num_threads(num_of_threads)
for (i = 0; i < num; i++)
{
    x = (i + 0.5)*step;
    S = S + 4.0 / (1.0 + x*x);
}
```

2. После введения параллелизма запустите программу. На консоли Вы увидите подсчитанное значение и время выполнения программы. Сделайте скрин консоли, сохраните его, назвав соответствующим образом. Запустите **Concurrency Analysis** инструмента **Amplifier XE** из панели инструментов **Visual Studio**. Во вкладке **Summary** отчета Вы должны увидеть цикл функции **par()**, использующий наибольшее время CPU. Нажав на него, Вы перейдете во вкладку **Bottom-up**. Оцените загруженность вычислителей, представленную на графике ниже. Сделайте скрин

вкладки **Bottom-up**, сохраните его, назвав соответствующим образом. Текущую версию программы и скрины добавьте в коммит и загрузите в **GitHub**.

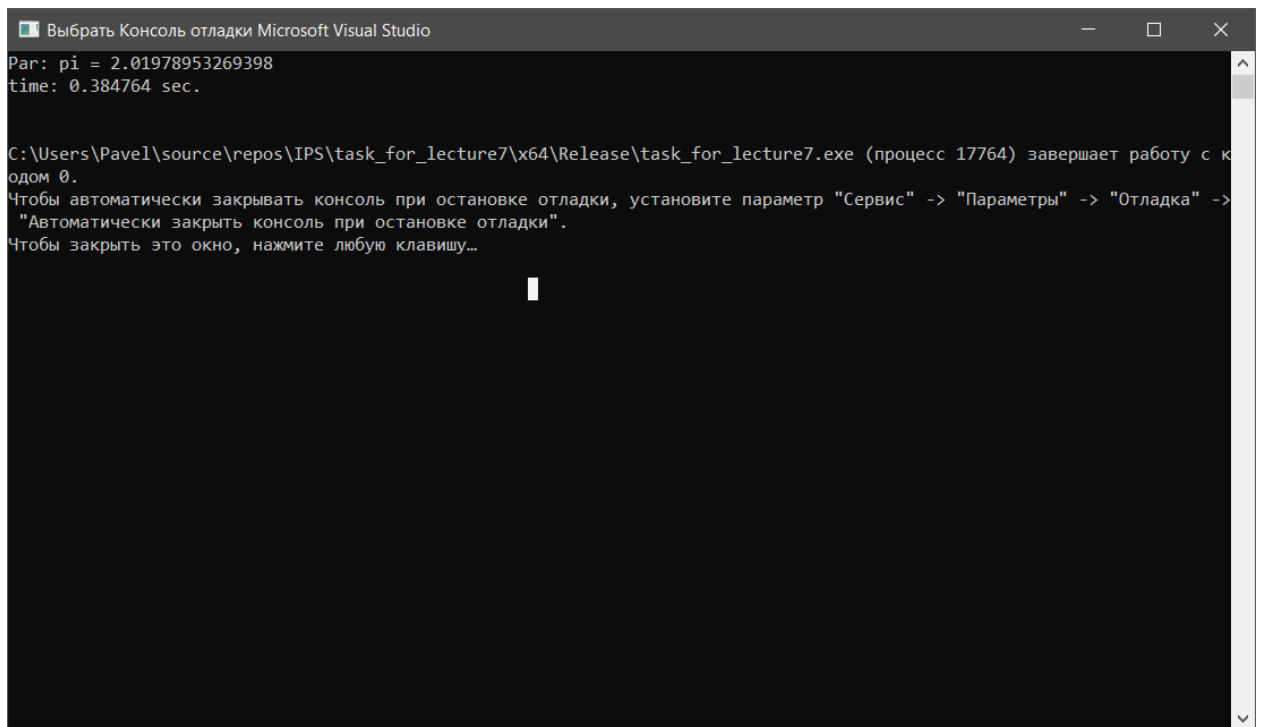


Рисунок 2 – Пример запуска программы после введения параллелизма

Как мы видим время работы программы уменьшилось, но вычисляемое значение π неверно.

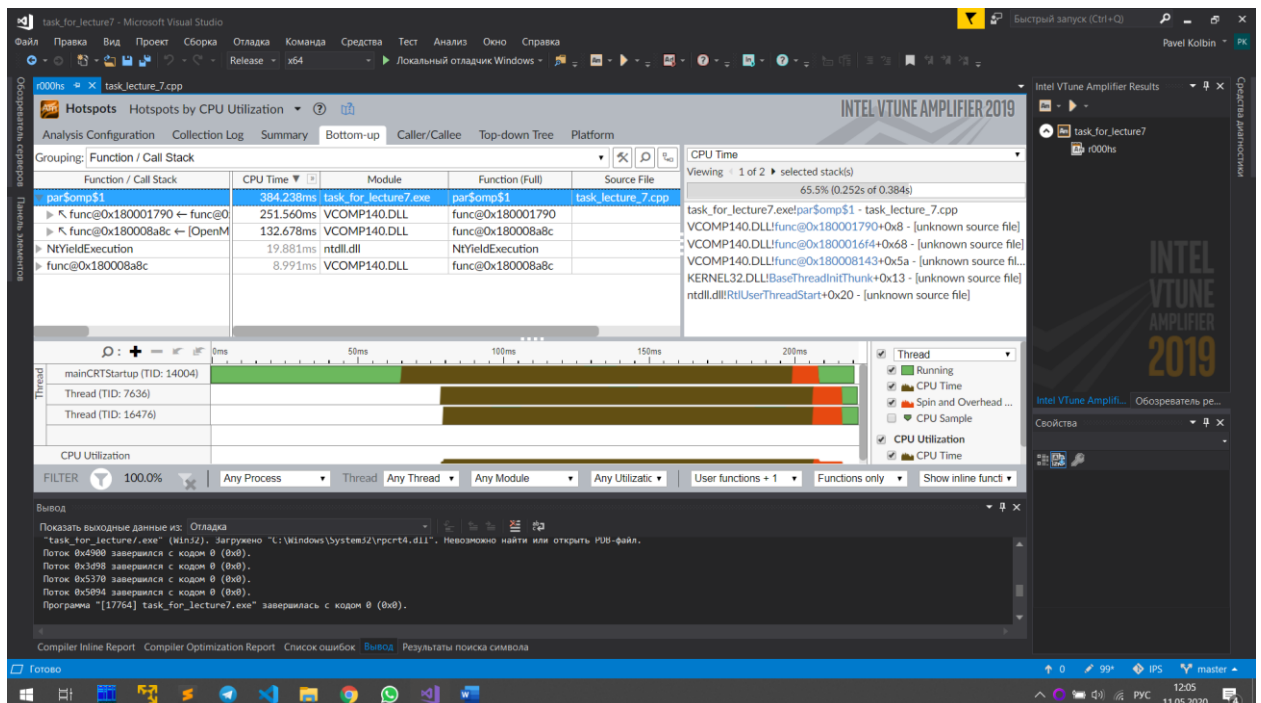


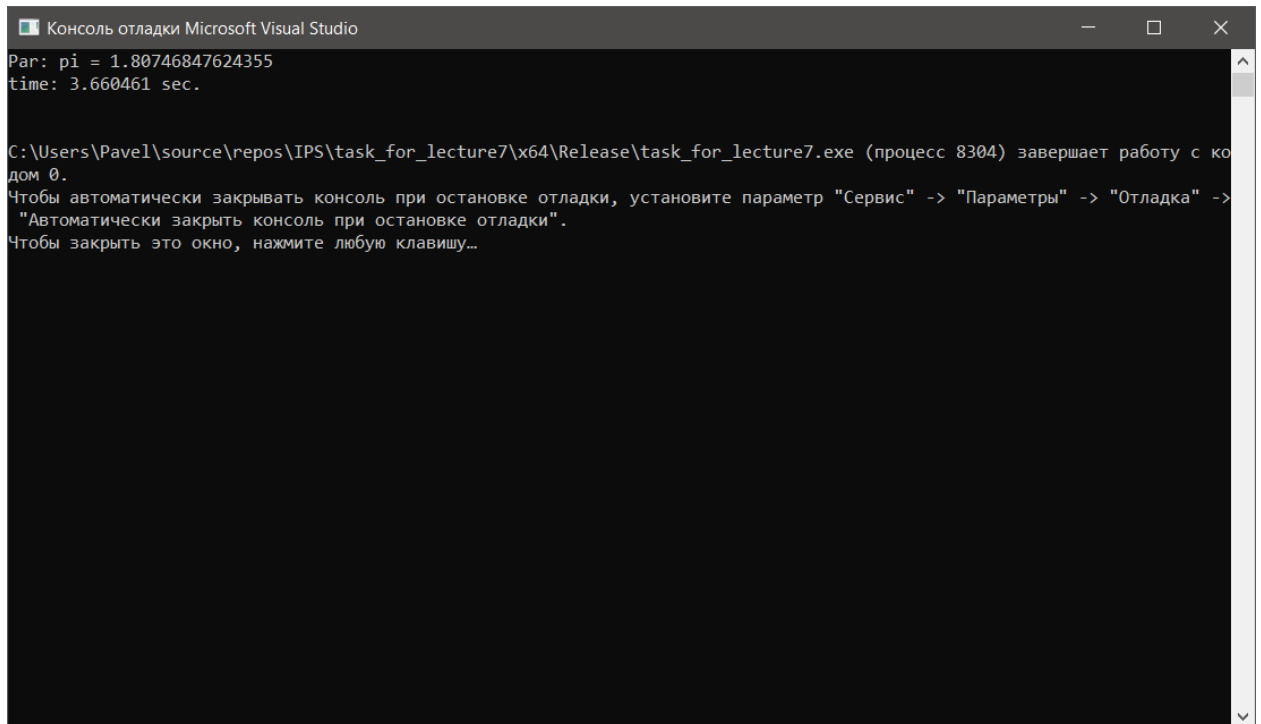
Рисунок 3 – Пример работы **Amplifier XE**

Во вкладке **Summary** отчета мы видим цикл функции **par()**, использующий наибольшее время CPU.

3. В функции **par()** в цикле по *i* от **0** до **num** после выражения $S = S + 4.0 / (1.0 + x*x)$; добавьте следующие 2 строки кода **#pragma omp atomic, inc++**; Пересоберите решение. Запустите программу, сделайте скрин консоли, сохраните его. Далее запустите **Concurrency Analysis**. Перейдя во вкладку **Summary** отчета, Вы увидите, что теперь наибольшее время затрачивается на выполнение новых двух добавленных строк кода. Чем Вы объясните такие изменения?

Добавим в код **#pragma omp atomic, inc++**;

```
#pragma omp parallel for num_threads(num_of_threads)
for (i = 0; i < num; i++)
{
    x = (i + 0.5)*step;
    S = S + 4.0 / (1.0 + x*x);
    #pragma omp atomic;
    inc++;
}
```



```
Консоль отладки Microsoft Visual Studio
Par: pi = 1.80746847624355
time: 3.660461 sec.

C:\Users\Pavel\source\repos\IPS\task_for_lecture7\x64\Release\task_for_lecture7.exe (процесс 8304) завершает работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, установите параметр "Сервис" -> "Параметры" -> "Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Чтобы закрыть это окно, нажмите любую клавишу...
```

Рисунок 4 – Пример запуска программы после добавление **#pragma omp atomic**

Как мы видим время сильно возросло, а результат все также вычисляется неправильно.

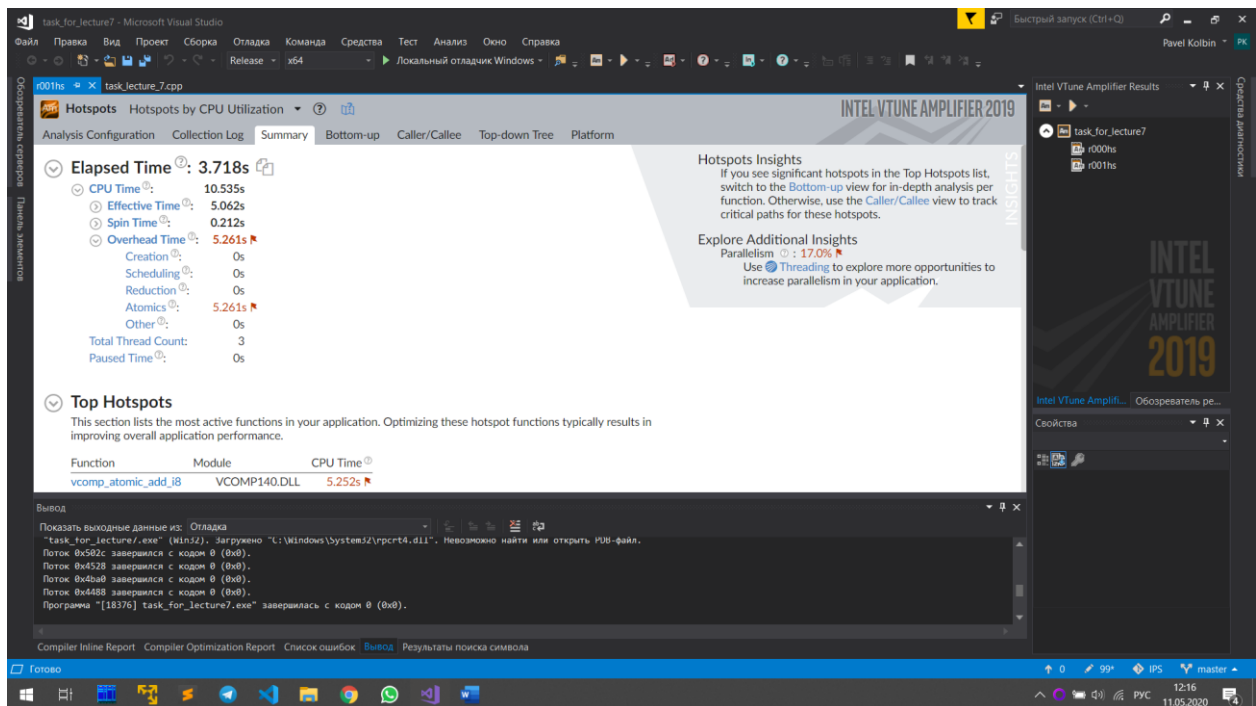


Рисунок 5 – Пример запуска **Amplifier XE** после добавление `#pragma omp atomic` вкладки **Summary**

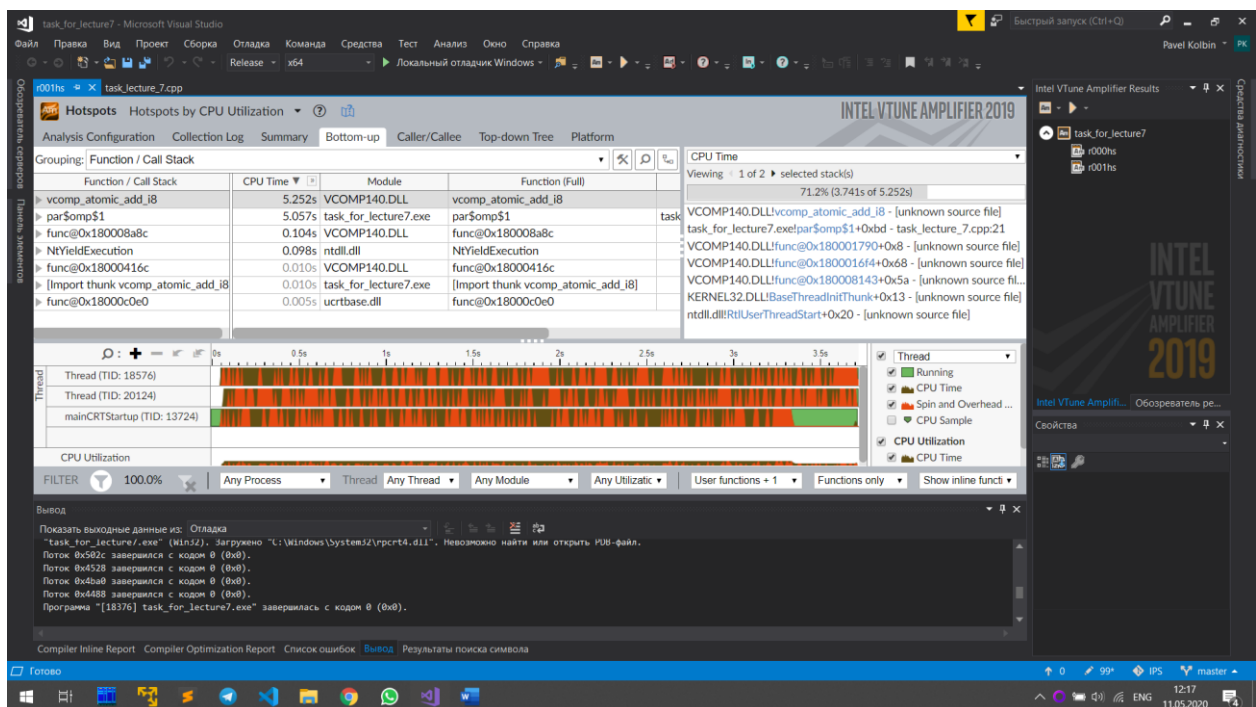


Рисунок 6 – Пример запуска **Amplifier XE** после добавление `#pragma omp atomic` вкладки **Bottom-up**

Действительно как мы видим основное время теперь на выполнение двух новых добавленных строк кода. Скорость работы приведенного кода будет ниже, чем скорость последовательного варианта. Во время работы алгоритма постоянно будут возникают блокировки, в результате чего практически вся работа ядер сведется к ожиданию. На практике использование этой директивы рационально при относительно редком обращении к общим переменным.

4. Замените строку **`#pragma omp atomic`** строкой **`#pragma omp critical`**. Пересоберите решение проекта, запустите программу. Сделайте скрин консоли, где отображено вычисленное значение и время выполнения программы.

Запустите **Concurrency Analysis**. Перейдя во вкладку **Summary** отчета Вы увидите изменения по сравнению с предыдущей версией программы. Чем Вы объясните такие изменения?

Далее, нажав по соответствующей строке отчета **Summary**, перейдите во вкладку **Bottom-up**. Проанализируйте загруженность вычислителей. сохраните скрин вкладки **Bottom-up**. Текущую версию программы и скрины добавьте в коммит и загрузите в **GitHub**.

Заменяли на `pragma omp critical`

```
#pragma omp parallel for num_threads(num_of_threads)
    for (i = 0; i < num; i++)
    {
        x = (i + 0.5)*step;
        S = S + 4.0 / (1.0 + x*x);
        //# pragma omp atomic
        #pragma omp critical
            inc++;
    }
```

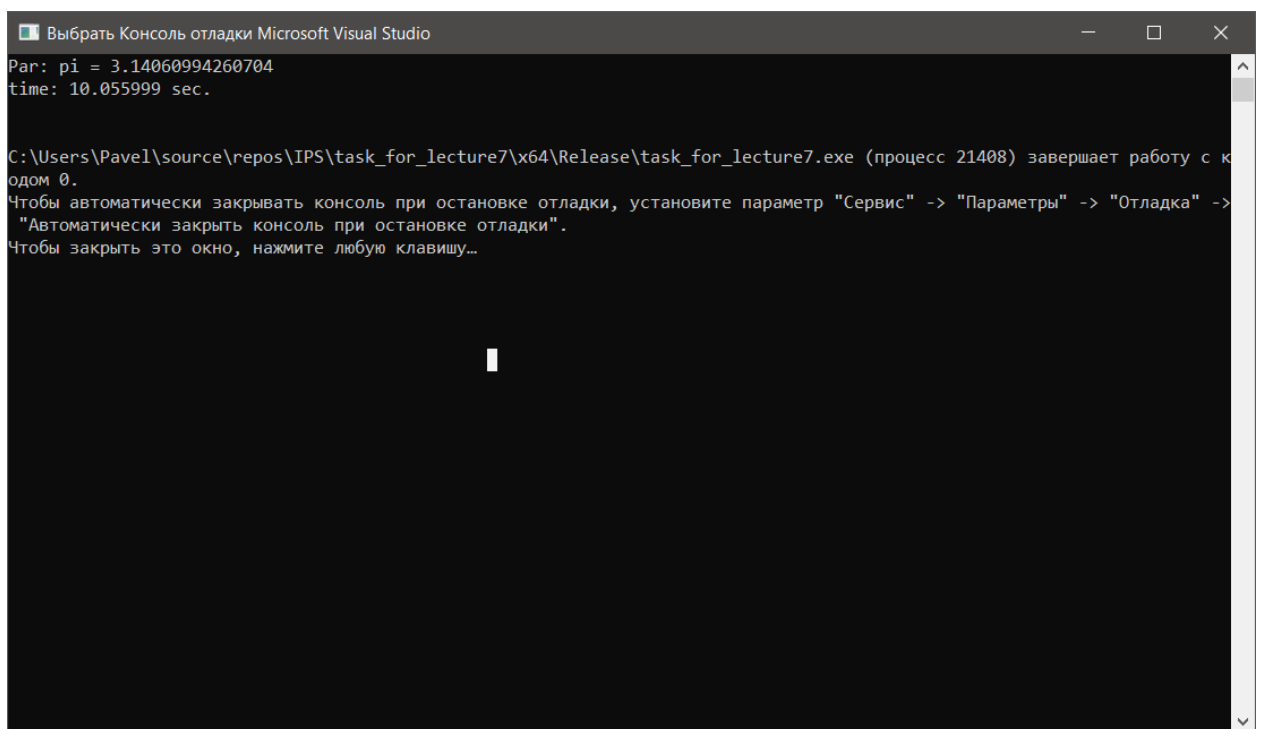


Рисунок 7 - Пример запуска программы после добавление `#pragma omp critical`

Как мы видим время выполнения программы возросло почти в 3 раза, но результат вычисления теперь верный.

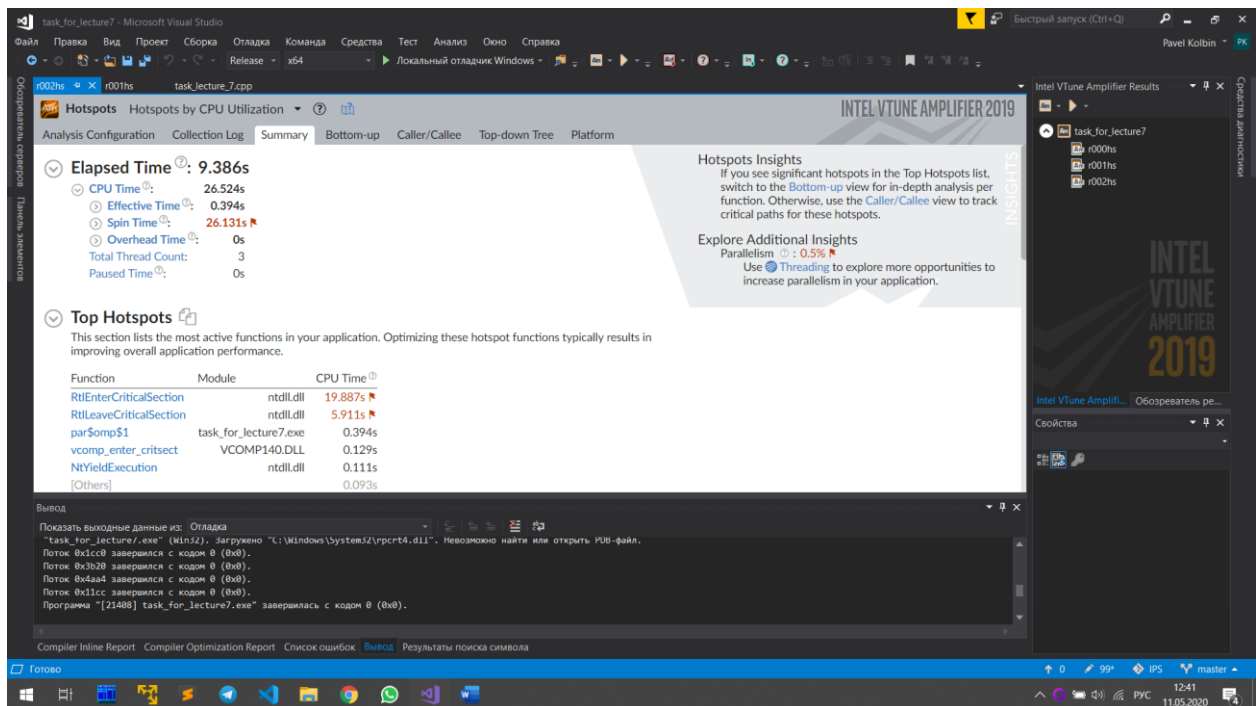


Рисунок 8 – Пример запуска **Amplifier XE** после добавление **#pragma omp critical** вкладки **Summary**

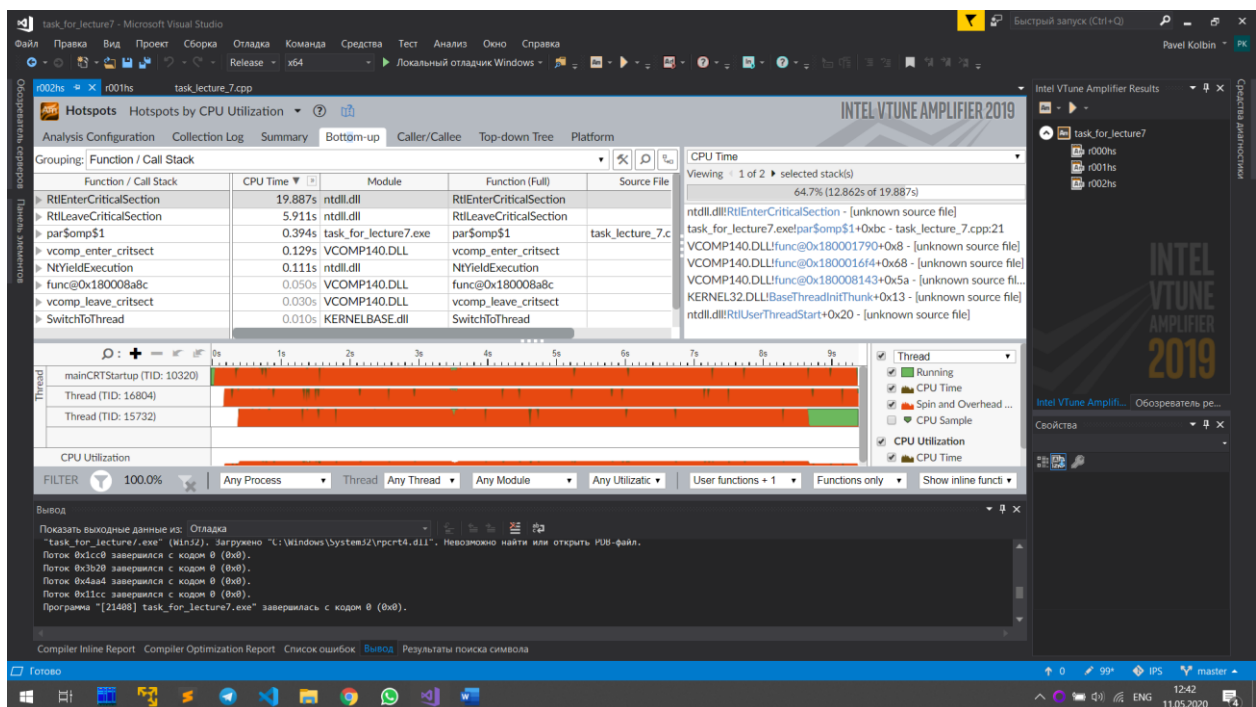


Рисунок 9 – Пример запуска **Amplifier XE** после добавление **#pragma omp critical** вкладки **Bottom-up**

С помощью директивы **critical** мы можем указать участок кода, который будет исполняться только одним потоком в один момент времени. Таким образом, гонок данных не возникает, но время выполнения программы сильно растет.

5. Замените строку **#pragma omp critical**. Введите в программу изменения: перед инкрементом переменной **inc** необходимо поставить вызов **omp_set_lock (&writelock)**, после него вызов **omp_unset_lock (&writelock)**. Пример правильного использования этих двух функций показан на изображении [init_lock_openmp.png](#). После введенных

изменений пересоберите решение, запустите программу. Сделайте скрин консоли. Запустите **Concurrency Analysis**. Во вкладке **Summary** отчета Вы должны увидеть, что в данном случае наибольшее время затрачивается на вызов функций **omp_set_lock (&writelock)** и **omp_unset_lock (&writelock)**. Нажав по соответствующей строке отчета **Summary**, Вы перейдете во вкладку **Bottom-up**. Проанализируйте загруженность вычислителей. Сделайте скрин вкладки **Bottom-up**, сохраните его.

Добавим функции синхронизации в код

```
omp_lock_t writelock;
omp_init_lock(&writelock);
#pragma omp parallel for num_threads(num_of_threads)
for (i = 0; i < num; i++)
{
    x = (i + 0.5)*step;
    S = S + 4.0 / (1.0 + x*x);
    omp_set_lock(&writelock);
    inc++;
    omp_unset_lock(&writelock);
}
omp_destroy_lock(&writelock);
```

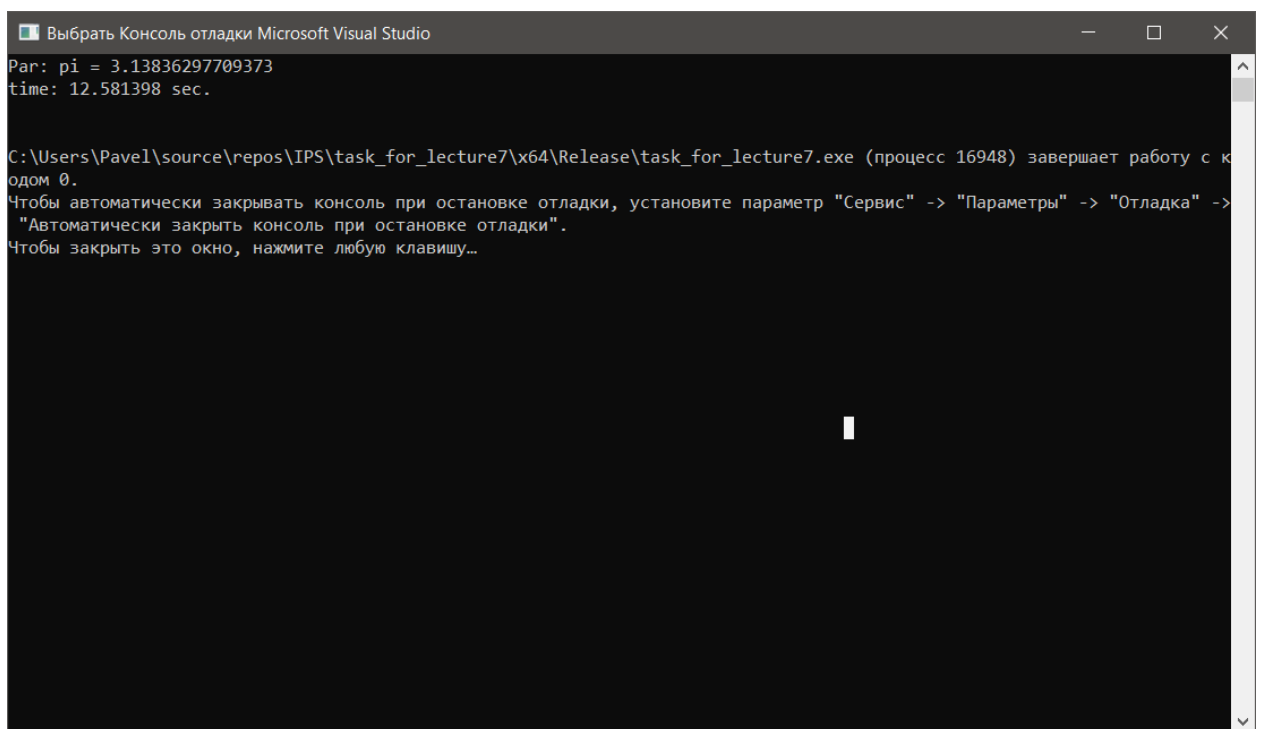


Рисунок 10 - Пример запуска программы после добавление механизма синхронизации

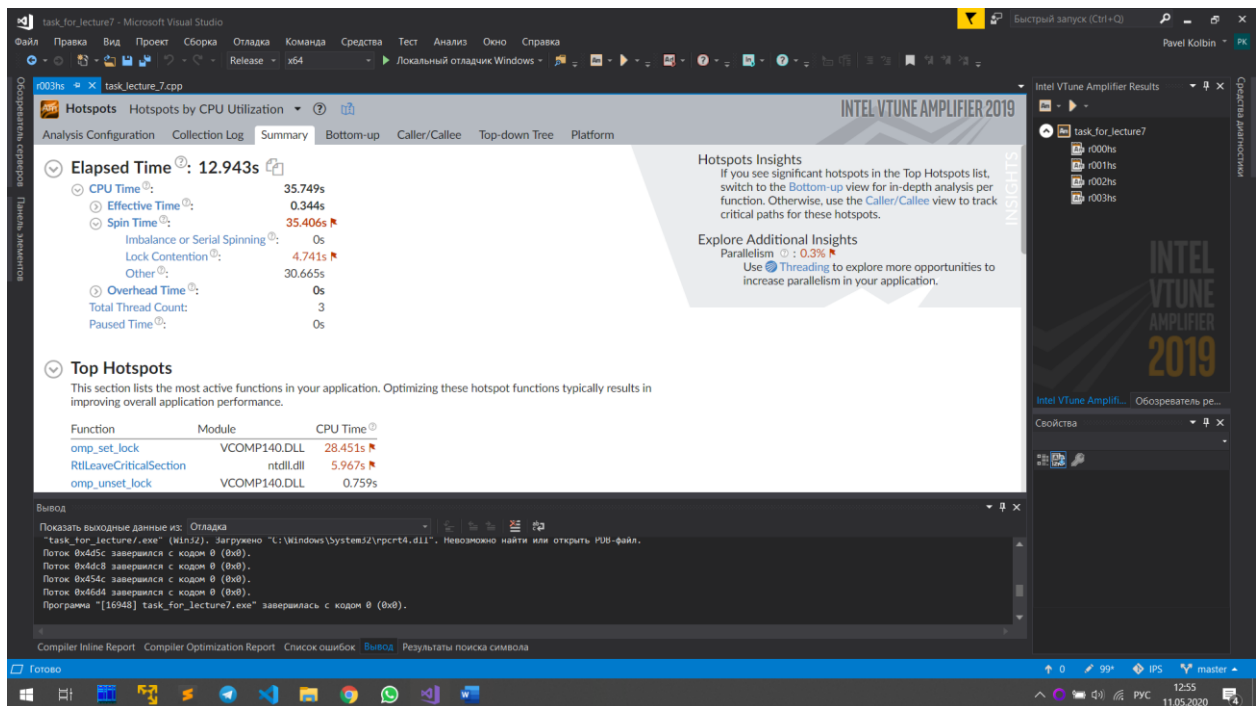


Рисунок 11 – Пример запуска **Amplifier XE** после добавление механизма синхронизации вкладка Summary

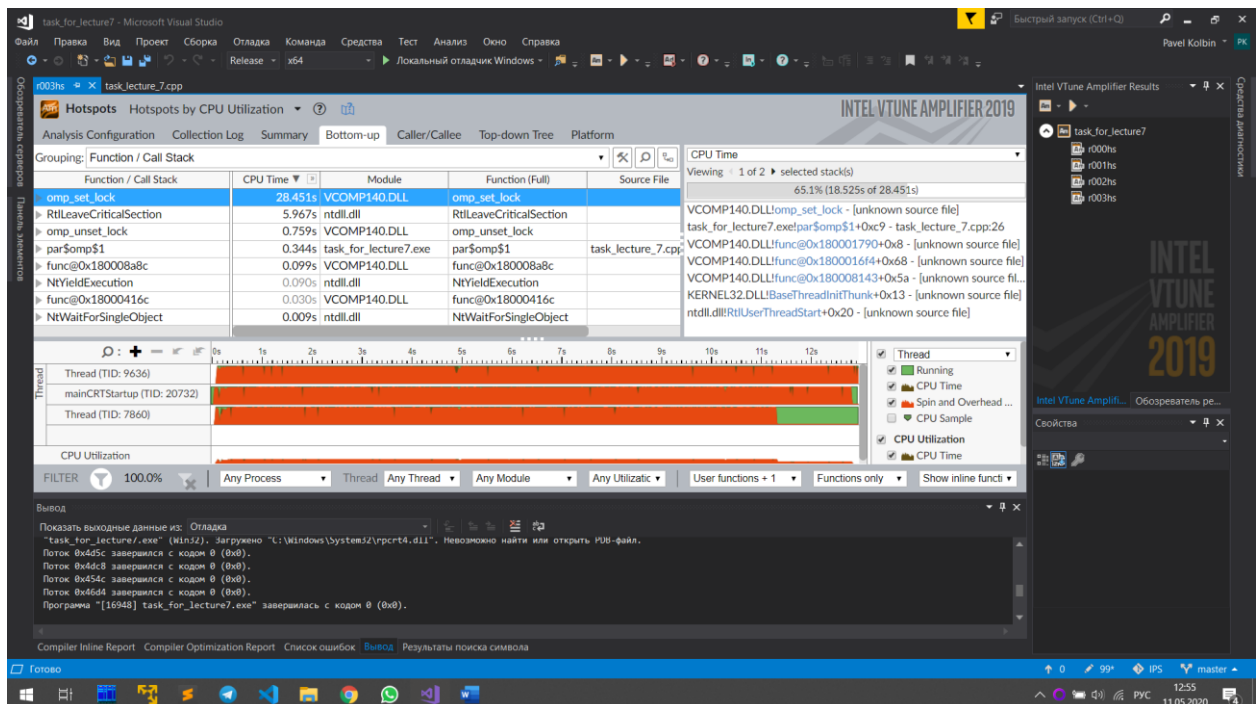


Рисунок 12 – Пример запуска **Amplifier XE** после добавление механизма синхронизации вкладка Bottom-up

Как видно на скриншотах результат вычисления правильный, но время выполнения также возросло. Теперь основное время затрачивается на операцию **set_lock**.

Вывод: таким образом, в рамках данной лабораторной работы я научился вводить параллелизм с помощью **OpenMP**, поработал с директивами **#pragma critical** и **#pragma atomic**, получил базовые знания по работе с механизмами синхронизации в **OpenMP**. Для анализа участков параллельного кода была использована утилита **Amplifier XE**. Согласно

полученным результатам операции для работы с потоками затрачивают достаточно много времени выполнения программы, так что их использование должно быть обдуманным и использоваться там, где это действительно необходимо, а именно в циклах с большим количеством итераций.