


Задание к занятию 7

Теверовский Михаил, ИВТ-11М

Используемые программы и характеристики ПК:

1. ПО: VisualStudio 2017 и Intel Parallel Studio 2019;
2. ОС: Windows 7 Professional
3. Процессор: Intel core i5-4200 CPU, 2.50 GHz, 2 ядра, 4 потока

Задание 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.

Результат:

1) Запустим представленный код task_lecture_7.cpp, предварительно настроив в свойствах проекта поддержку стандарта OpenMP:

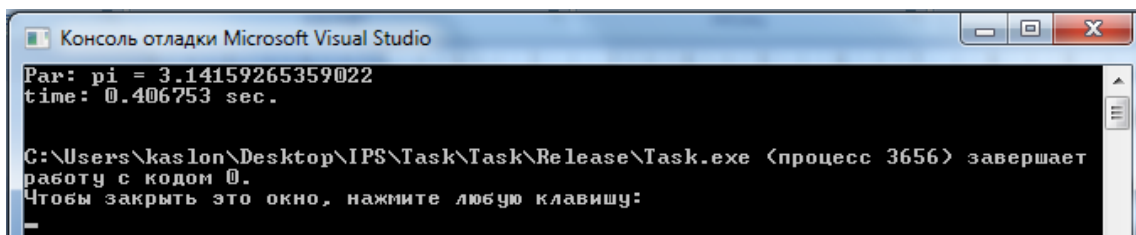


Рисунок 1. – Скриншот результата выполнения задания 1

2) Установим количество рабочих процессов равным 3:

Для этого используем:

```
#pragma omp parallel for num_threads(num_of_threads)
```

Полностью обновлённый код выглядит так:

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

long long num = 100000000;
double step;
double par(void)
{
    int num_of_threads = 3;
    long long inc = 0;
    long long i = 0;
    double x = 0.0;
    double pi;
    double S = 0.0;
    step = 1.0 / (double)num;
    double t = omp_get_wtime();
```

```
#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);
}
t = omp_get_wtime() - t;
pi = step * S;
printf("Par: pi = %.14f\n", pi);
return t;
}

int main()
{
    printf("time: %f sec.\n\n", par());return 0;
}
```

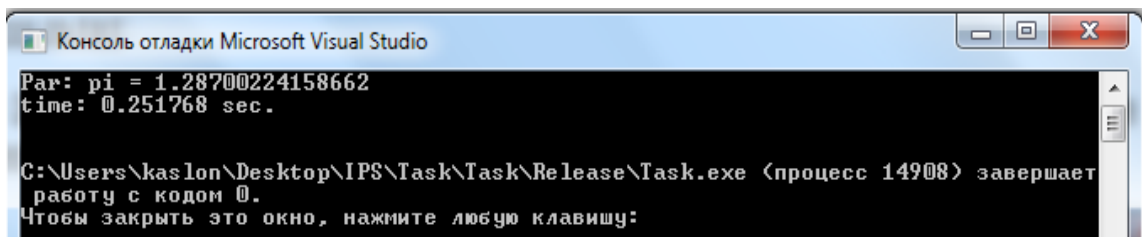


Рисунок 2. – Скриншот результата выполнения задания 1

Анализ полученных результатов:

Сравним полученные значения:

	Первоначальный код	Программа с введением количества рабочих процессов
Значение π	3.14159265359022	1.28700224158662
Время выполнения	0.406753	0.251768

Таблица 1. – Результаты выполнения задания 1

Таким образом – вычисляемое значение неверное, а время выполнения уменьшилось при введении количества рабочих процессов, равного 3.

Задание 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 загрузженность 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*.

1) Запуск программы и её результат приведены в пункте Задание 1

2) Запуск *Concurrency Analysis* инструмента *Amplifier XE*:

Результат:

Во вкладке Summary виден цикл, использующий наибольшее время CPU. Скриншот ниже:

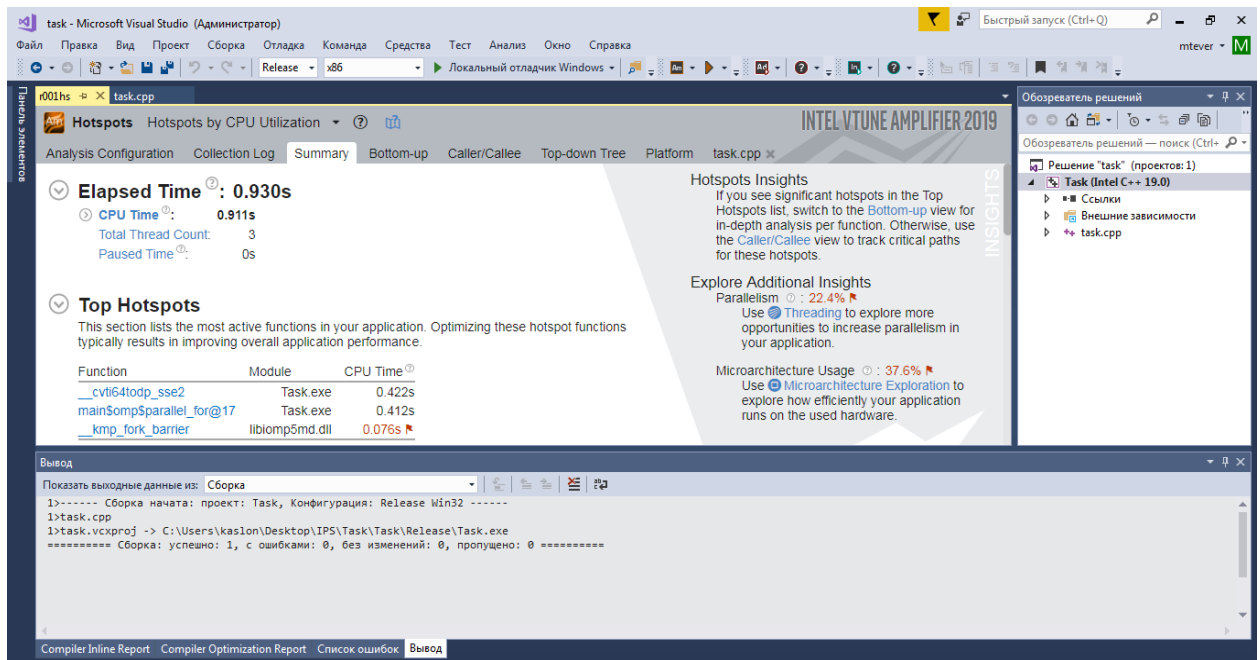


Рисунок 3. – Скриншот результата выполнения задания 2. Summary

Скриншот вкладки Bottom-up приведён на Рисунке 4 ниже:

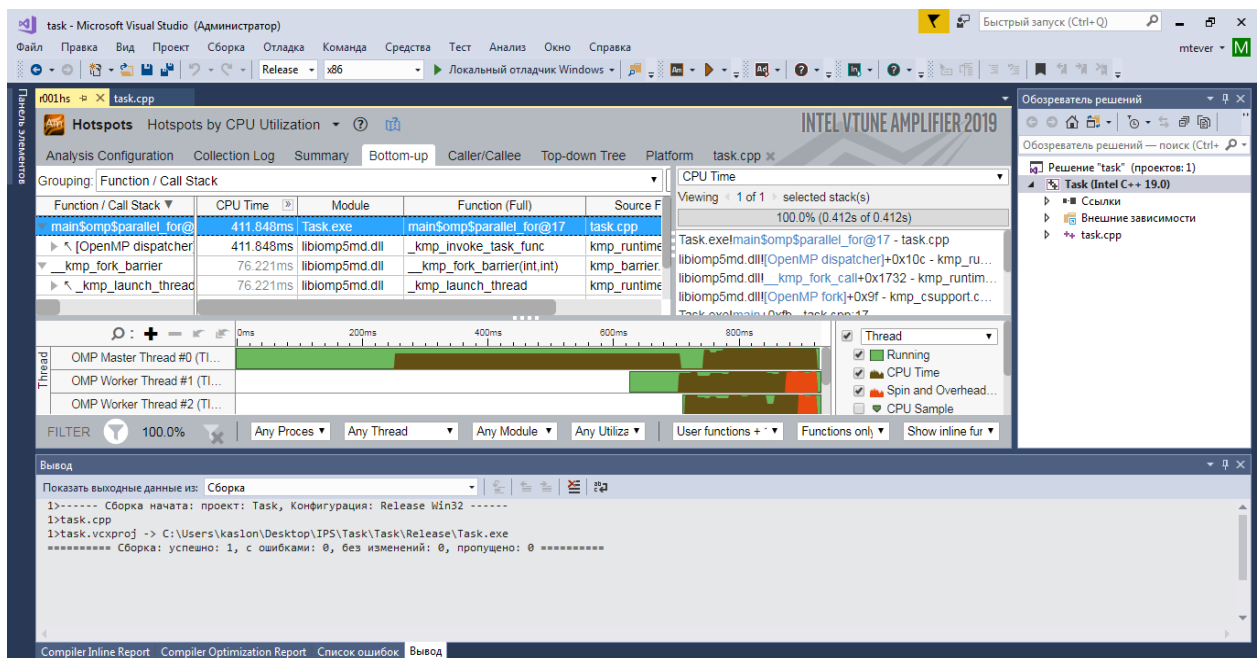


Рисунок 4. – Скриншот результата выполнения задания 2. Bottom-up

Задание 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?

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

1) **Введём в код программы 2 строки:**

```
#pragma omp atomic
inc++;
```

Полностью обновлённый код теперь выглядит следующим образом:

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

long long num = 100000000;
double step;
double par(void)
{
    int num_of_threads = 3;
    long long inc = 0;
    long long i = 0;
    double x = 0.0;
    double pi;
    double S = 0.0;
    step = 1.0 / (double)num;
    double t = omp_get_wtime();

#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++;
    }

    t = omp_get_wtime() - t;
    pi = step * S;
    printf("Par: pi = %.14f\n", pi);
    return t;
}

int main()
{
    printf("time: %f sec.\n\n", par());return 0;
}
```

Результат:

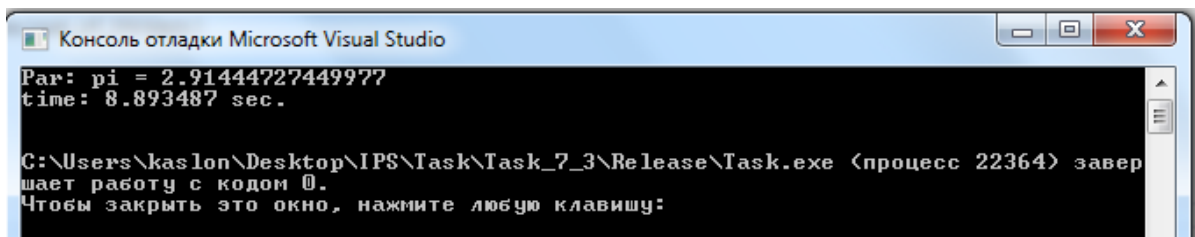


Рисунок 5. – Скриншот результата выполнения задания 3

Анализ полученных результатов:

- 1) Значение π всё также неверно, но уже ближе к действительному.
- 2) Время выполнения возросло на порядок.

2) Запустим Concurrency Analysis:

Результат:

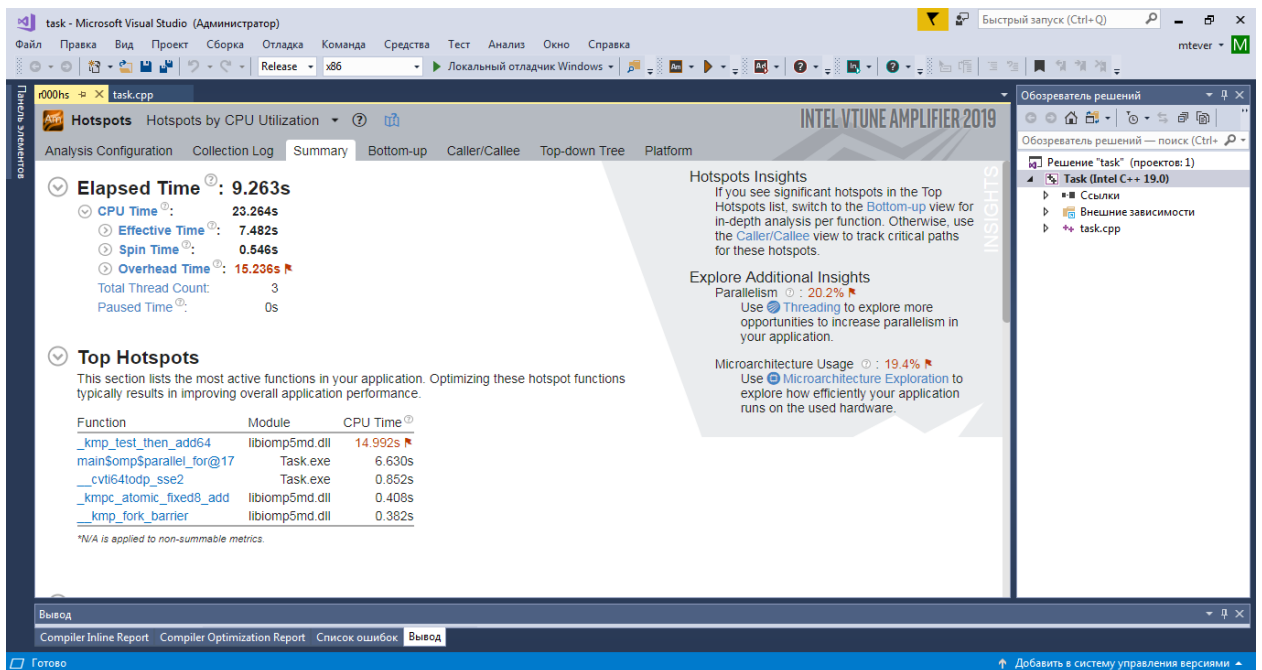


Рисунок 6. – Скриншот результата выполнения задания 3. Summary

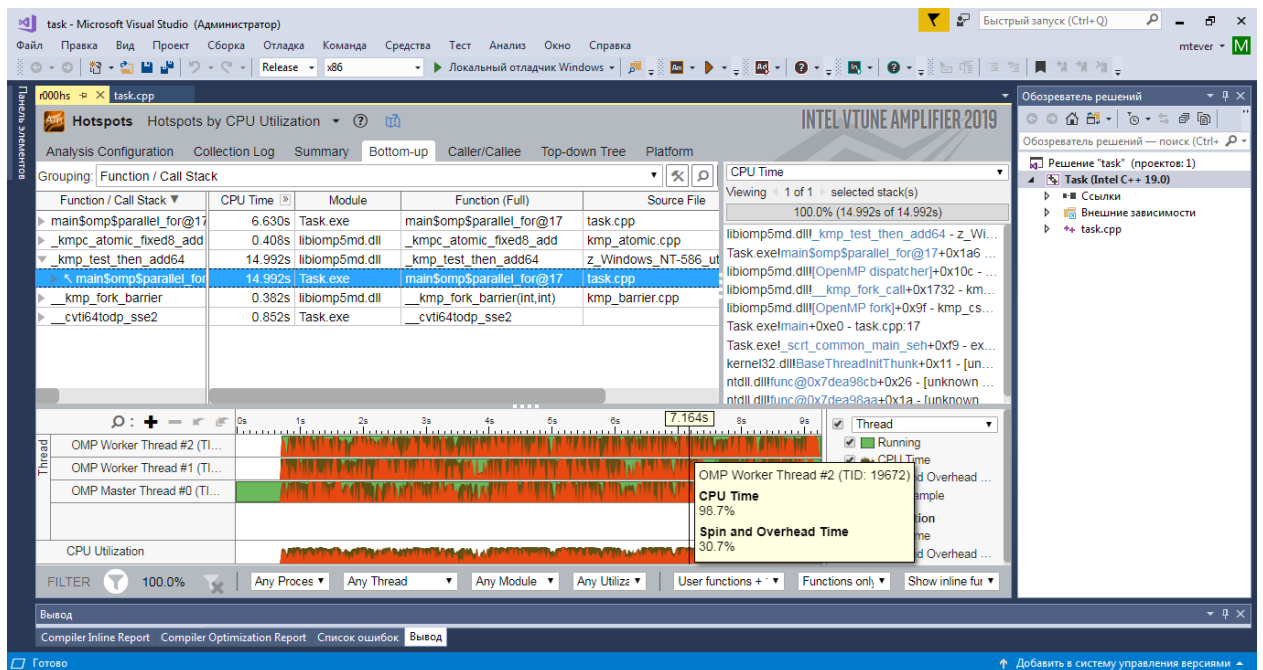


Рисунок 7. – Скриншот результата выполнения задания 3. Bottom-up

Анализ полученных результатов:

Время выполнение увеличилось с 400 миллисекунд до 14 секунд – это также было заметно при запуске программы. В итоге получится, что выбор данного варианта параллелизма будет выполняться медленнее последовательного кода, что абсолютно невыгодно.

Задание 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*.

1) Заменим строку `#pragma omp atomic` строкой `#pragma omp critical`:

Теперь код выглядит следующим образом:

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

long long num = 100000000;
double step;
double par(void)
{
    int num_of_threads = 3;
    long long inc = 0;
    long long i = 0;
    double x = 0.0;
    double pi;
```

```

double S = 0.0;
step = 1.0 / (double)num;
double t = omp_get_wtime();

#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 critical
    inc++;
}

t = omp_get_wtime() - t;
pi = step * S;
printf("Par: pi = %.14f\n", pi);
return t;
}

int main()
{
    printf("time: %f sec.\n\n", par());return 0;
}

```

Результат:

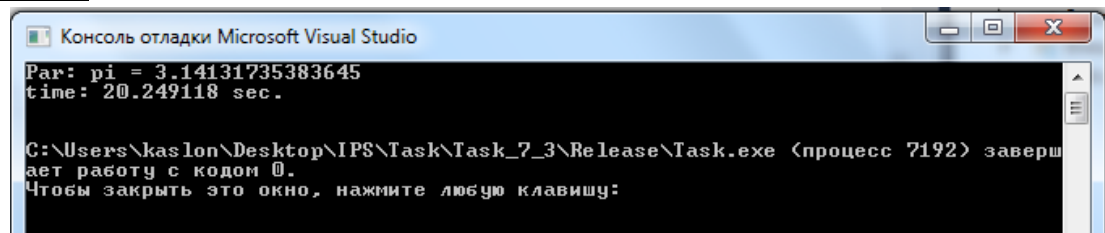


Рисунок 8. – Скриншот результата выполнения задания 4

Анализ полученных результатов:

	Первоначальный код	Код из задания 4
Значение π	3.14159265359022	3.14131735383645
Время выполнения	0.406753	20.249118

Таблица 2. – Результаты выполнения задания 4

- 1) Значение π всё ещё неидеально, но приближается к верному – ошибка в 4 разряде дробной части
- 2) Время выполнения всё также на порядки выше

2) Запустим Concurrency Analysis:

Результат:

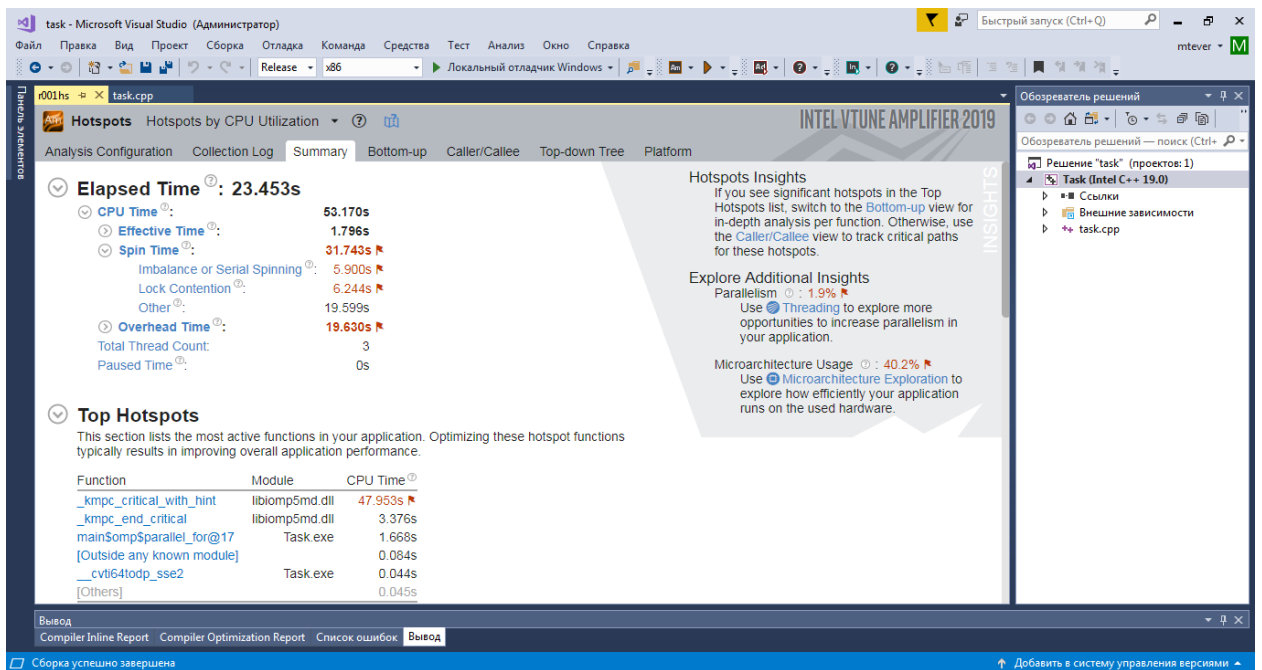


Рисунок 9. – Скриншот результата выполнения задания 4. Summary

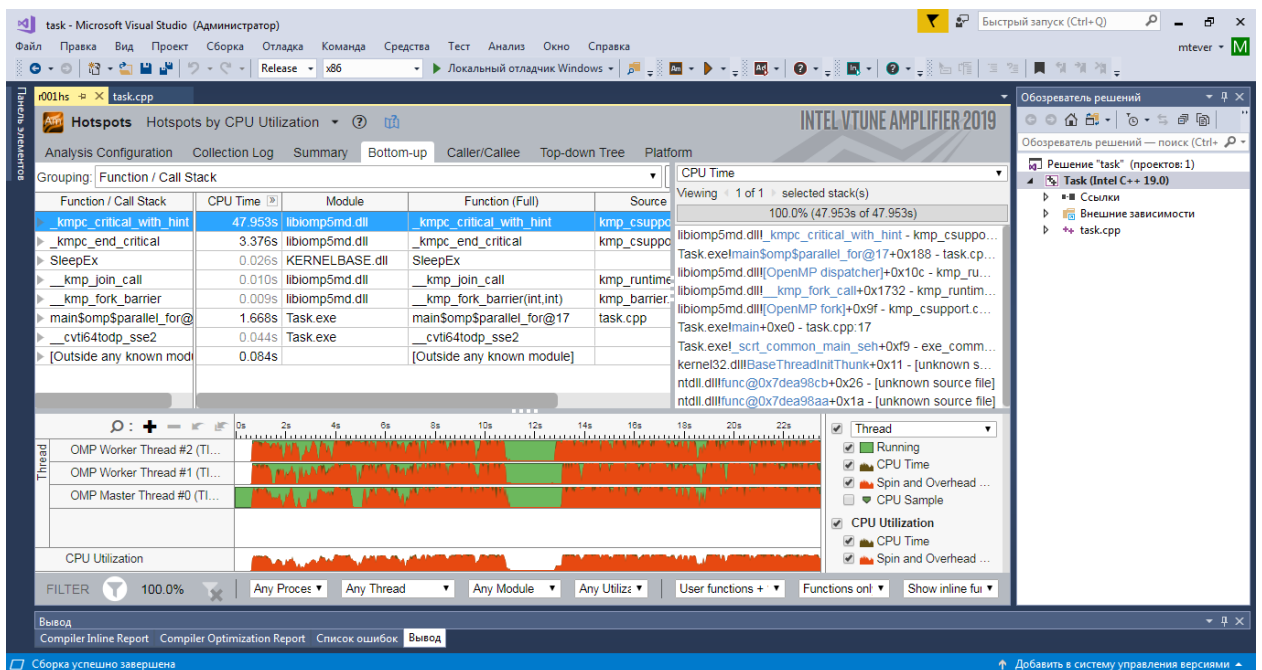


Рисунок 10. – Скриншот результата выполнения задания 4. Bottom-up

Анализ полученных результатов:

При помощи **critical** указываем участок кода, для выполнения которого в один момент времени будет задействован только один поток. Вследствие этого время выполнения программы увеличивается

Задание 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 загрузенность вычислителей. Сделайте скрин вкладки *Bottom-up*, сохраните его.

1) Изменённый участок кода:

```
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);
```

Теперь код программы выглядит следующим образом:

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

long long num = 100000000;
double step;
double par(void)
{
    int num_of_threads = 3;
    long long inc = 0;
    long long i = 0;
    double x = 0.0;
    double pi;
    double S = 0.0;
    step = 1.0 / (double)num;
    double t = omp_get_wtime();

    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);
t = omp_get_wtime() - t;
pi = step * S;
printf("Par: pi = %.14f\n", pi);
return t;
}

int main()
{
```

```
}  
    printf("time: %f sec.\n\n", par());return 0;  
}
```

Результат:

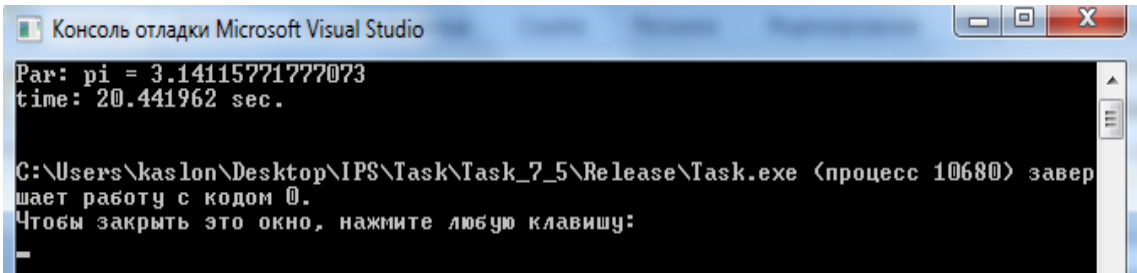


Рисунок 11. – Скриншот результата выполнения задания 5

Анализ полученных результатов:

	Первоначальный код	Код из задания 5
Значение pi	3.14159265359022	3.14115771777073
Время выполнения	0.406753	20.441962

Таблица 3. – Результаты выполнения задания 5

Результаты соизмеримы с полученными в пункте «Задание 4»

- 1) Значение pi всё ещё неидеально, но приближается к верному – ошибка всё также в 4 разряде дробной части
- 2) Время выполнения всё также на порядки выше

2) Запустим *Concurrency Analysis*:

Результат:

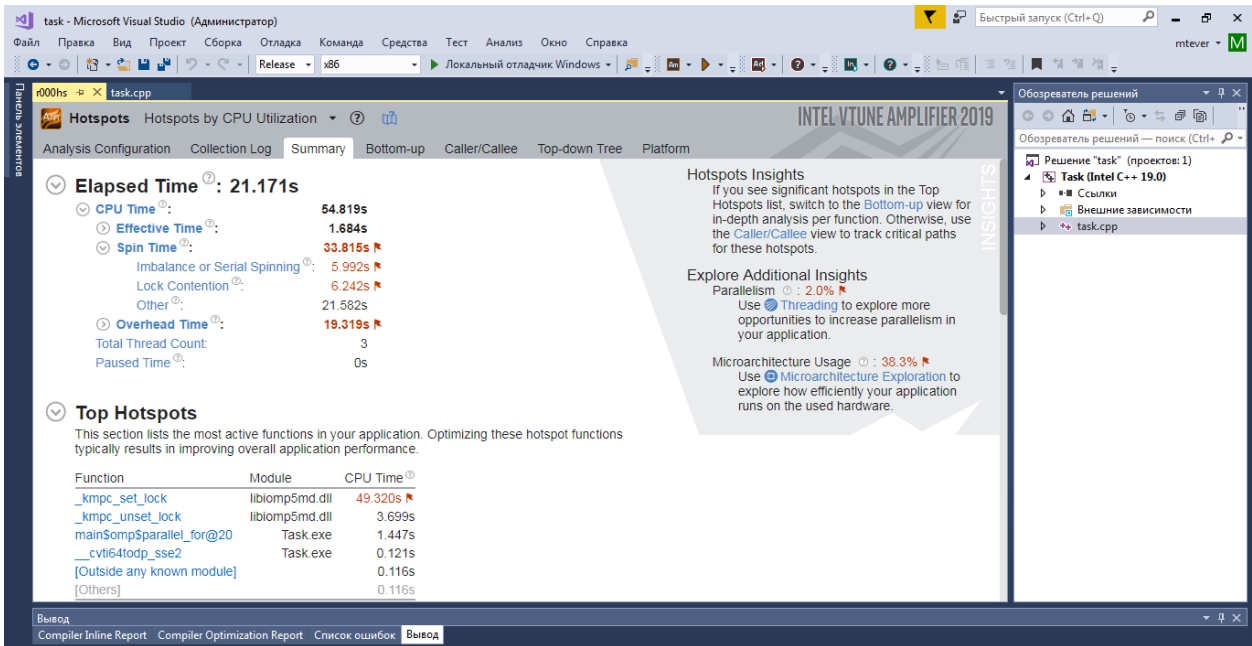


Рисунок 12. – Скриншот результата выполнения задания 5. Bottom-up

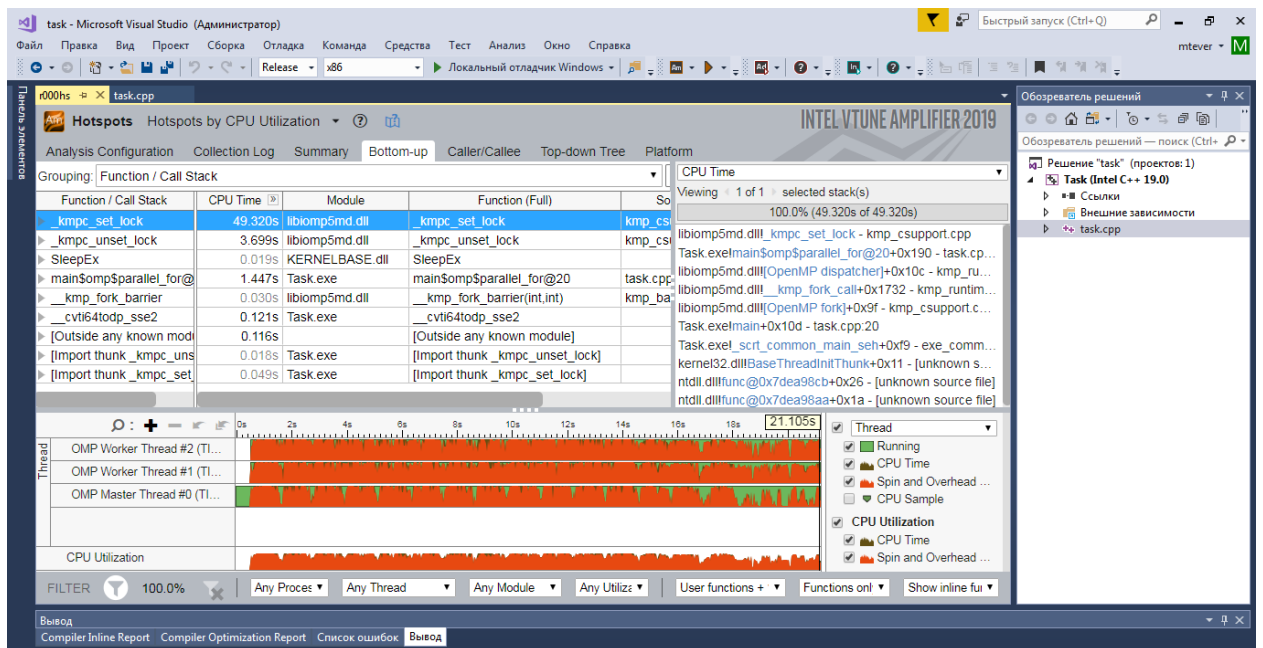


Рисунок 13. – Скриншот результата выполнения задания 5. Bottom-up

Анализ полученных результатов:

Время выполнения сильно возросло – судя по Amplifier XE много времени затрачивается на новую теперь операцию set_lock.

При запуске выполнения программы также наблюдается долгая задержка выдачи результатов.

Выводы:

По результатам работы стоит отметить, что при работе с потоками при помощи OpenMT затрачивается достаточно много времени выполнения программы – следовательно, его использование, как и были сделаны выводы в предыдущих работах, возможно в программах с большим количеством данных.

В рамках задания к занятию 7 мной были изучены:

- 1) Изучение методов введения параллелизма с помощью OpenMT;
- 2) Такие директивы, как:
 - 1) #pragma critical
 - 2) #pragma atomic
- 3) Функции блокировки и разблокировки потока: omp_set_lock и omp_unset_lock
- 4) Повторена работа с Amplifier XE