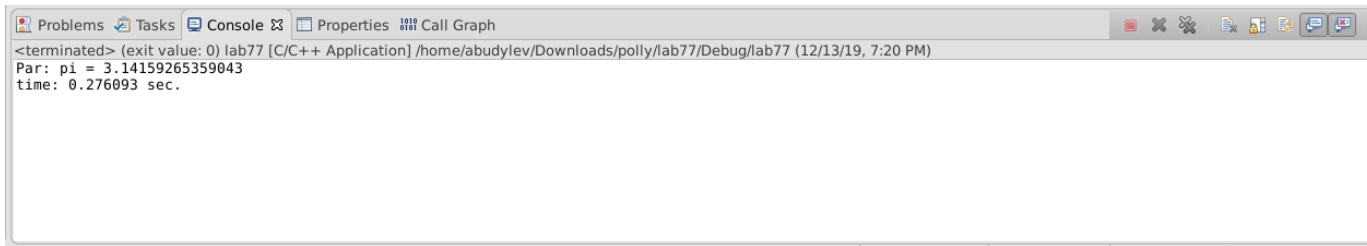


Отчет к занятию 7.

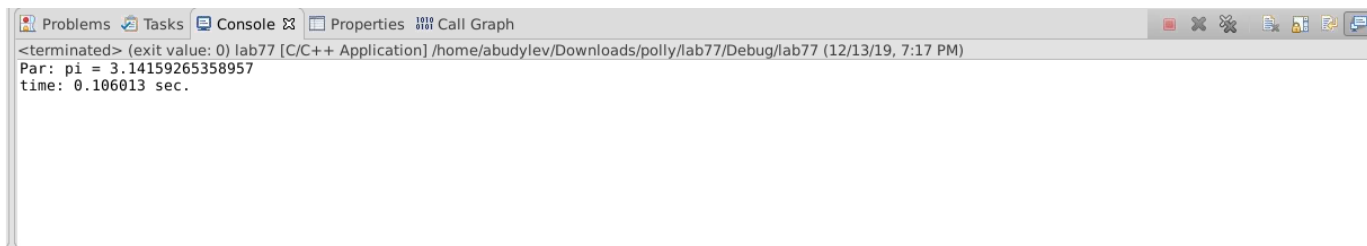
1. Разберите последовательную программу по вычислению определенного интеграла `task_lecture_7.cpp`. Введите в нее параллелизм с помощью OpenMP. Установите количество рабочих процессов равным 3, для этого используйте оператор `num_threads(num_of_threads)`. Не забудьте настроить в свойствах проекта поддержку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

Работа программы без параллелизма (скрин консоли)



```
Problems Tasks Console Properties Call Graph
<terminated> (exit value: 0) lab77 [C/C++ Application] /home/abudylev/Downloads/polly/lab77/Debug/lab77 (12/13/19, 7:20 PM)
Par: pi = 3.14159265359043
time: 0.276093 sec.
```

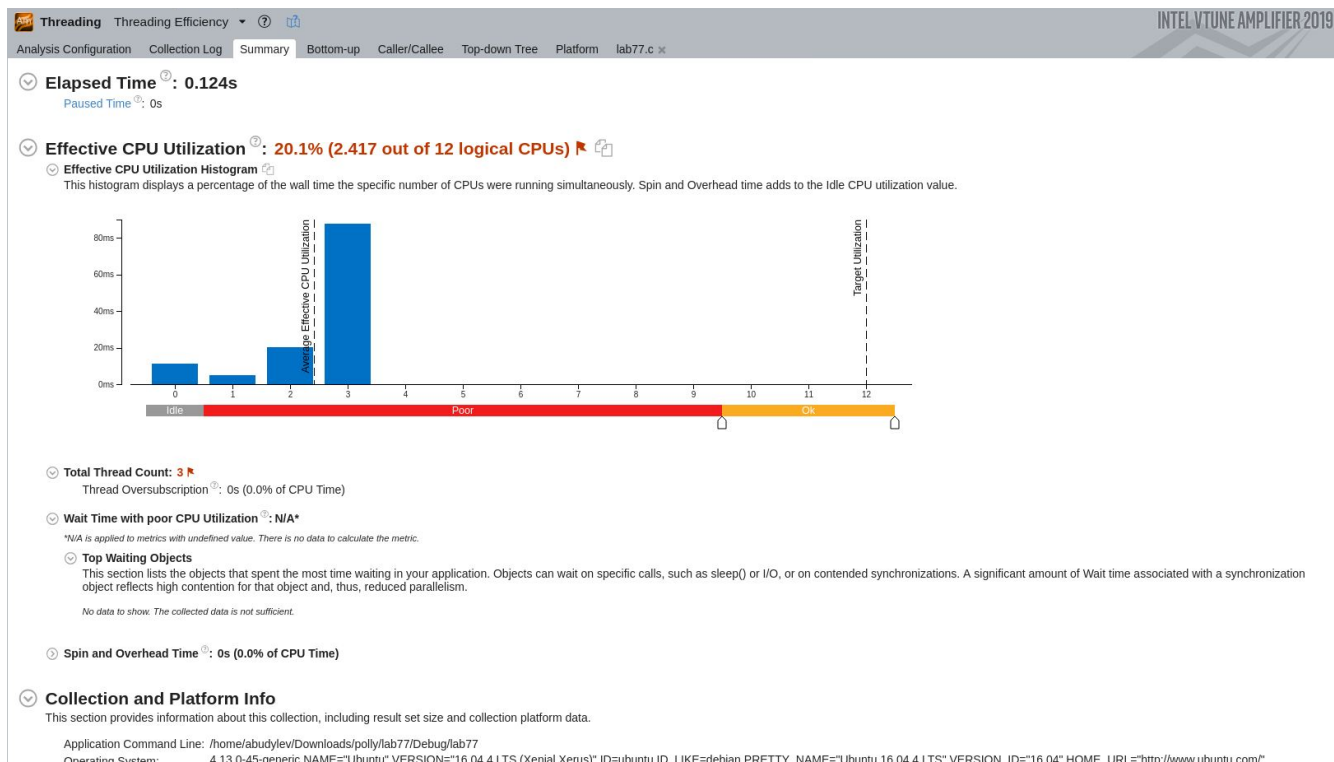
Работа параллельной программы. 3 процесса. (скрин консоли)



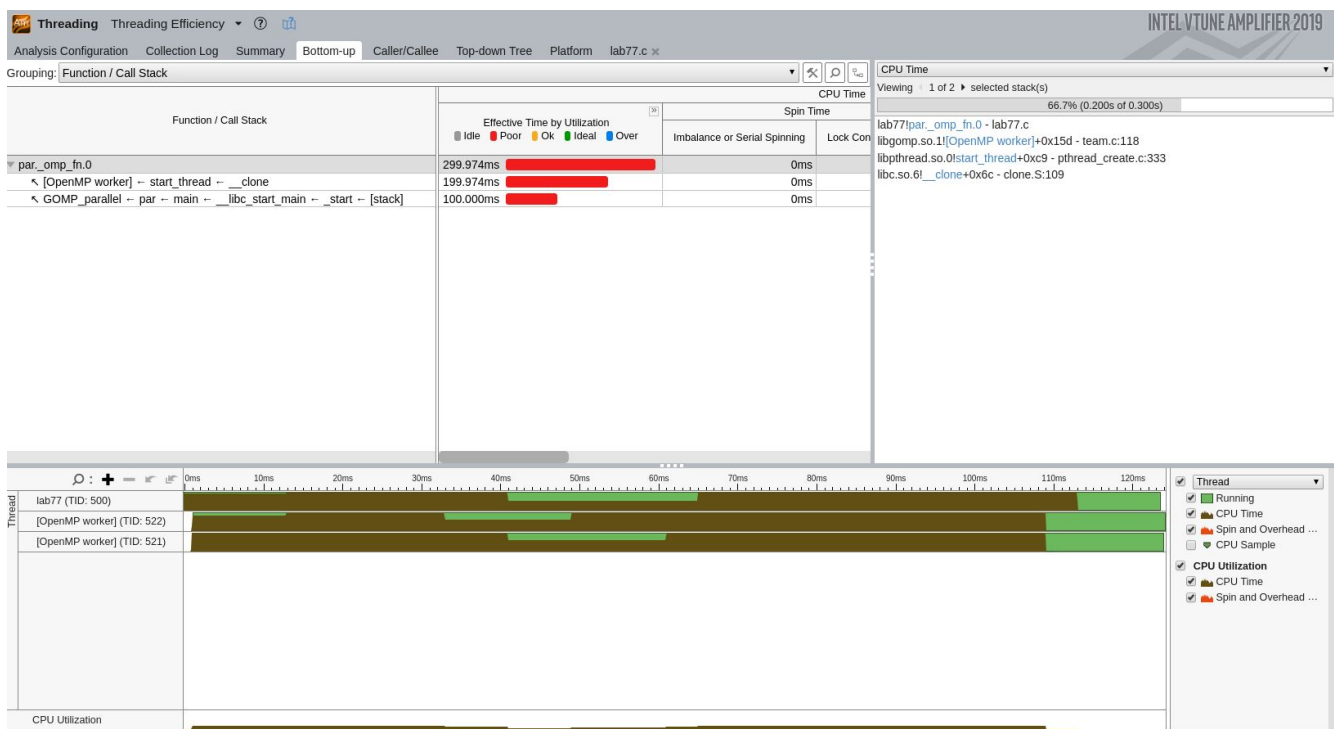
```
Problems Tasks Console Properties Call Graph
<terminated> (exit value: 0) lab77 [C/C++ Application] /home/abudylev/Downloads/polly/lab77/Debug/lab77 (12/13/19, 7:17 PM)
Par: pi = 3.14159265358957
time: 0.106013 sec.
```

Как видим, есть ускорение времени работы программы.

Вкладка Summary

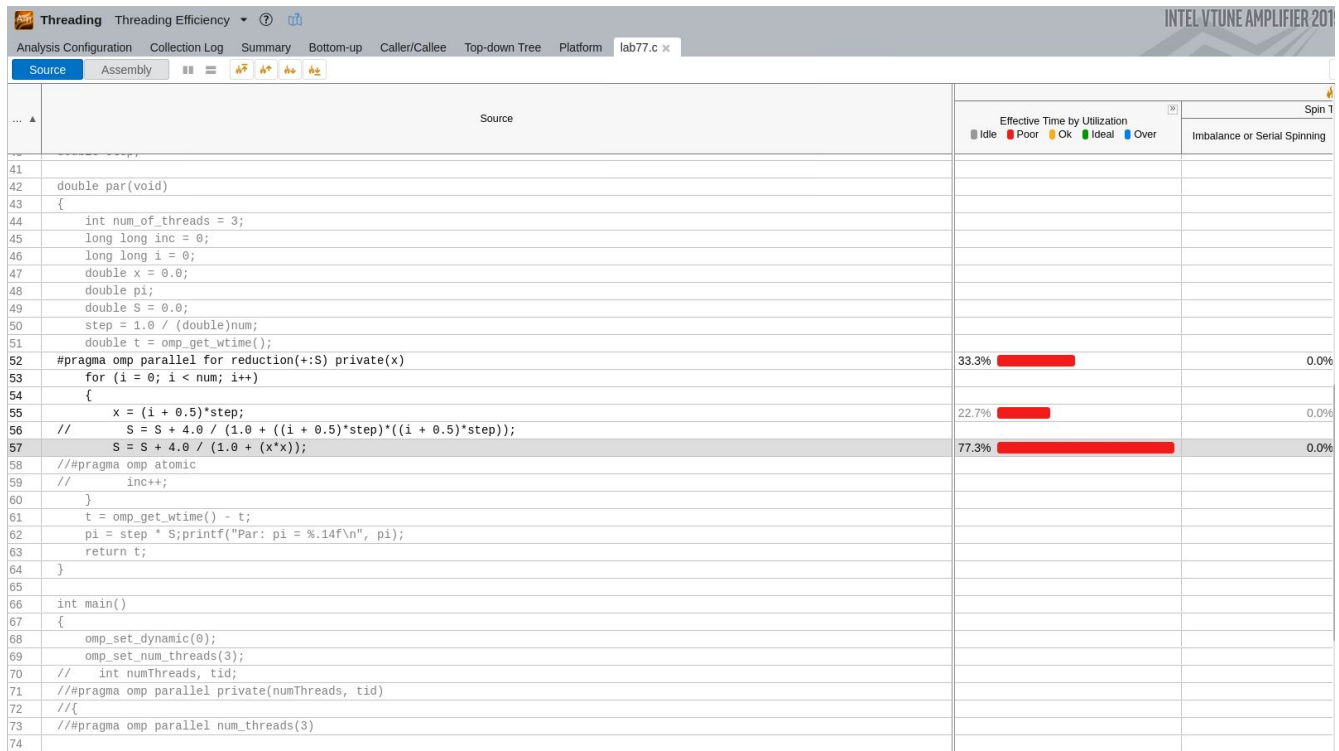


Вкладка Bottom-up



Загруженность вычислителей: вычислители загружены более-менее равномерно и используются достаточно эффективно.

Время (в процентах от общего) выполнения строчек кода (для сравнения со следующим заданием).



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

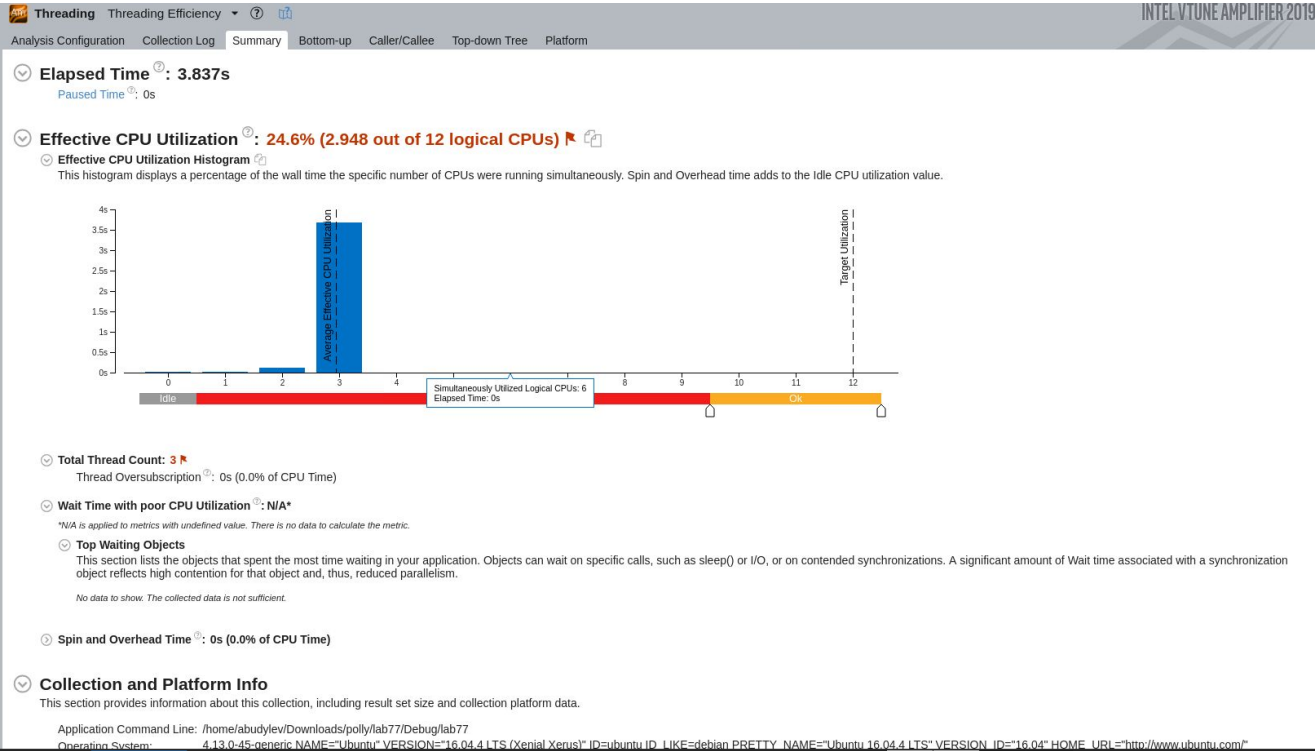
Работа программы с двумя добавленными строчками.



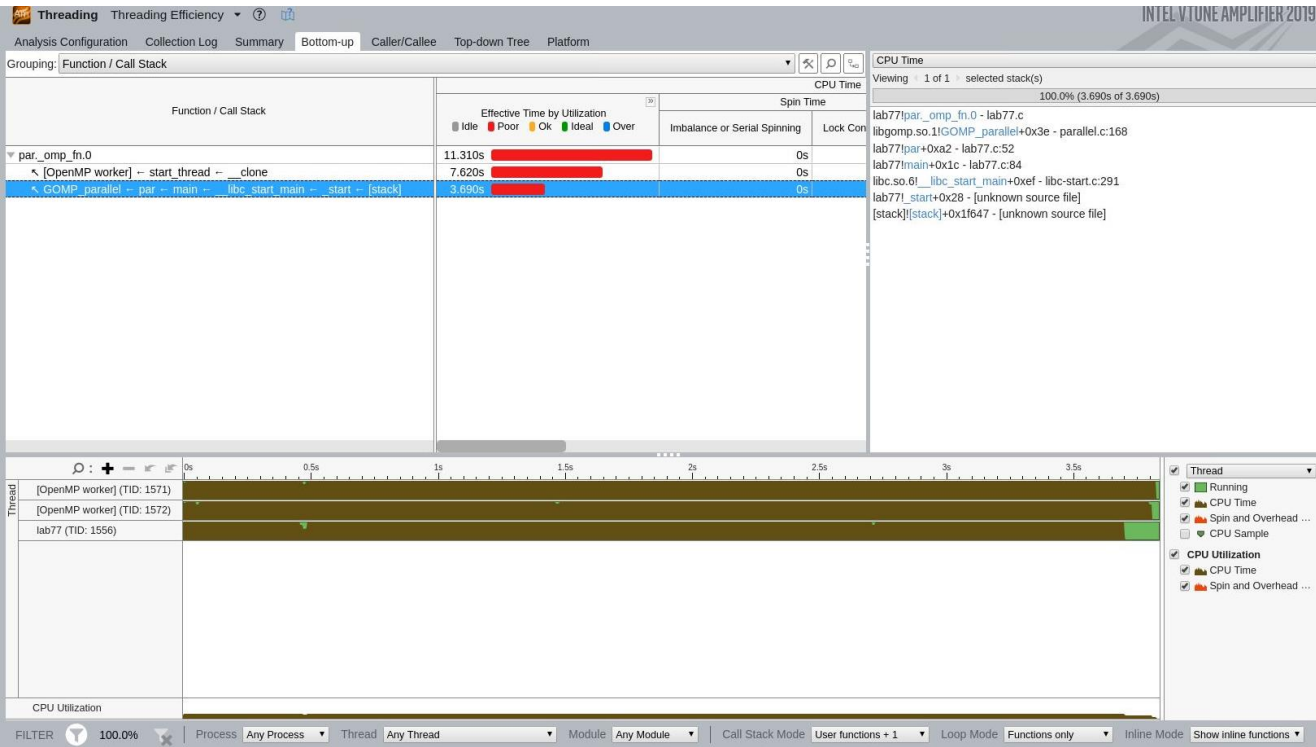
Программа работает дольше. По анализу программы видно, что эффективность распараллеливания не поменялась, а все время затрачивается на выполнение добавленных строк кода.

Операция `atomic` выполняется без прерывания, что создает общую нагрузку на процессор. Атомарные операции потребляют больше времени на выполнение.

Вкладка Summary

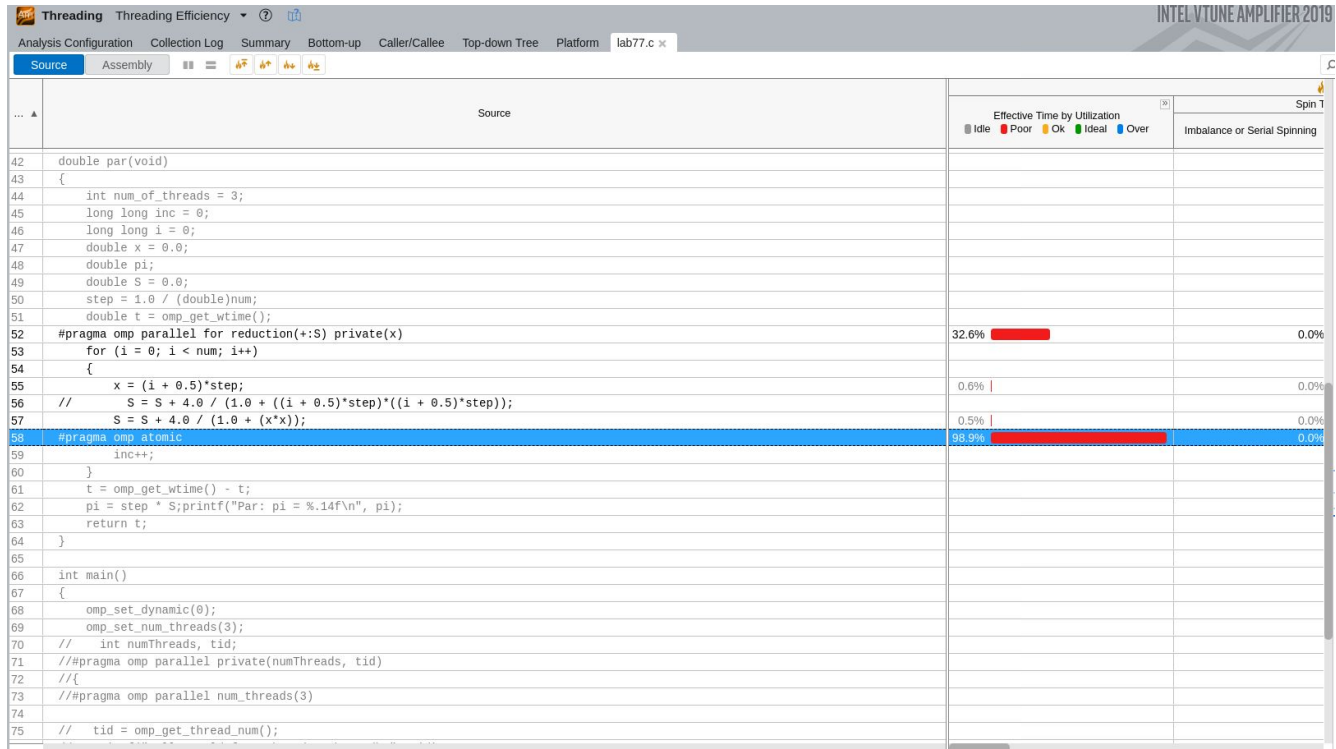


Вкладка Bottom-up



Загруженность вычислителей: аналогично предыдущему заданию.

Время (в процентах от общего) выполнения строчек кода.



Сравнив с результатами из прошлого задания, видим, что все время уходит на операцию atomic.

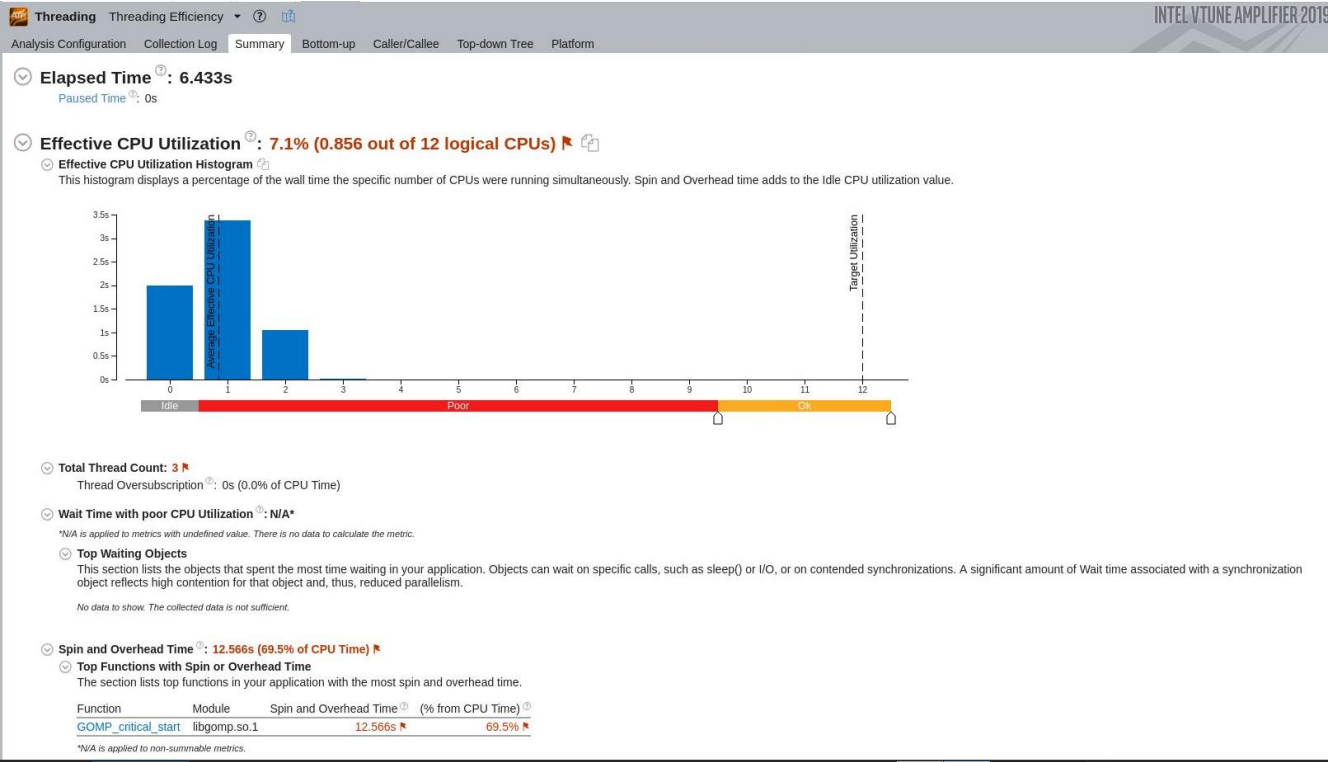
4. Замените строку `#pragma omp atomic` строкой `#pragma omp critical`. Пересоберите решение проекта, запустите программу. Сделайте скрин консоли, где отображено вычисленное значение и время выполнения программы. Запустите Concurrency Analysis. Перейдя во вкладку Summary отчета Вы увидите изменения по 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.

Программа после изменения atomic на critical.

```
<terminated> (exit value: 0) lab77 [C/C++ Application] /home/abudylev/Downloads/polly/lab77/Debug/lab77 (12/13/19, 7:29 PM)
Par: pi = 3.14159265358957
time: 7.204646 sec.
```

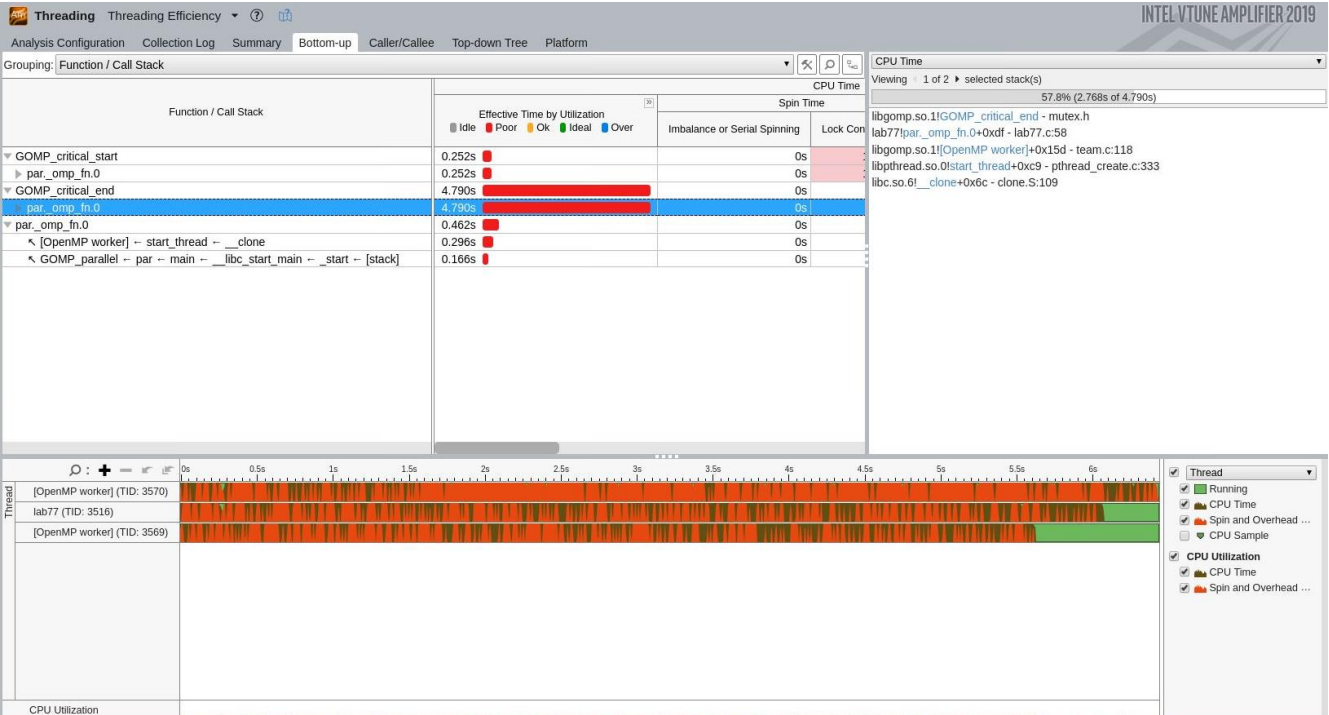
Время работы программы увеличилось еще сильнее. Потому что критическая операция предотвращает множественный одновременный доступ к определенному сегменту кода. Поток получает доступ к критической операции лишь в том случае, когда эта критическая операция не обрабатывается другим потоком.

Вкладка Summary

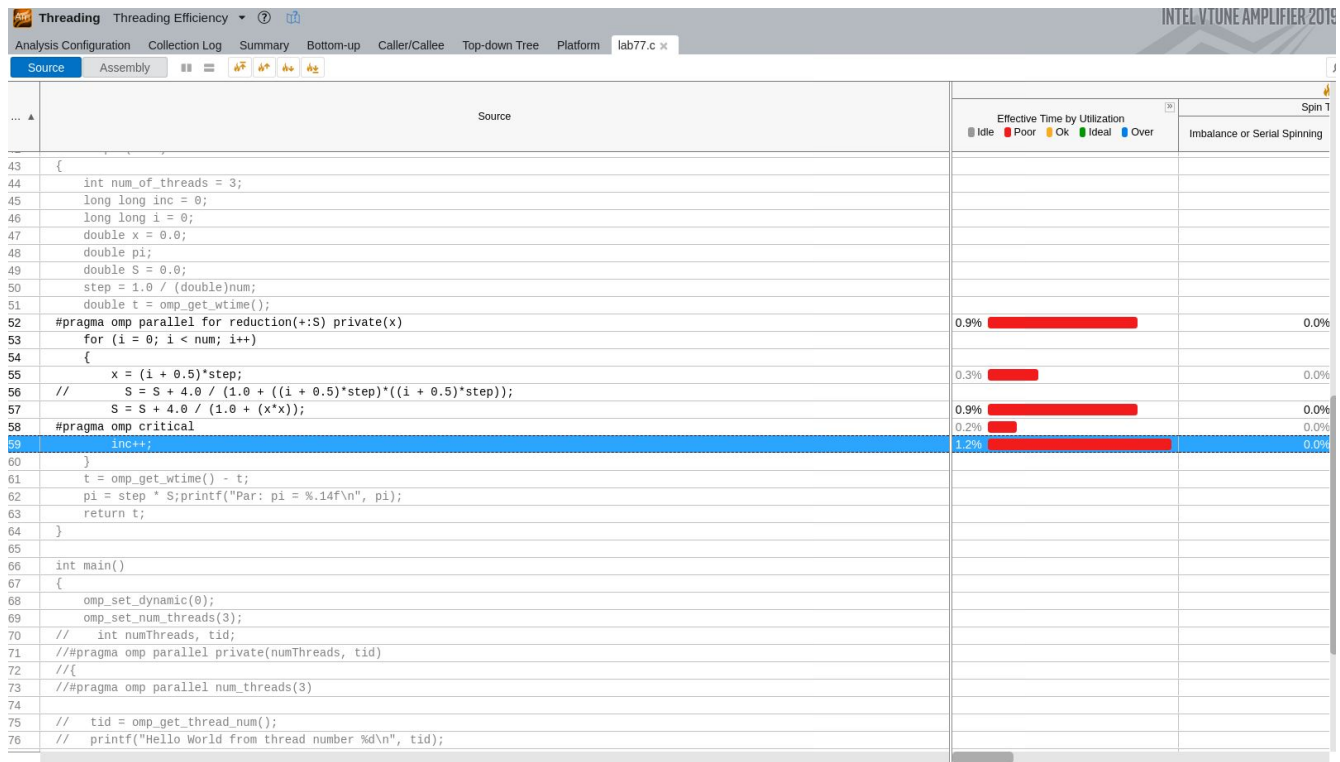


Эффективность упала примерно в три раза.

Вкладка Bottom-up



Время (в процентах от общего) выполнения строчек кода.

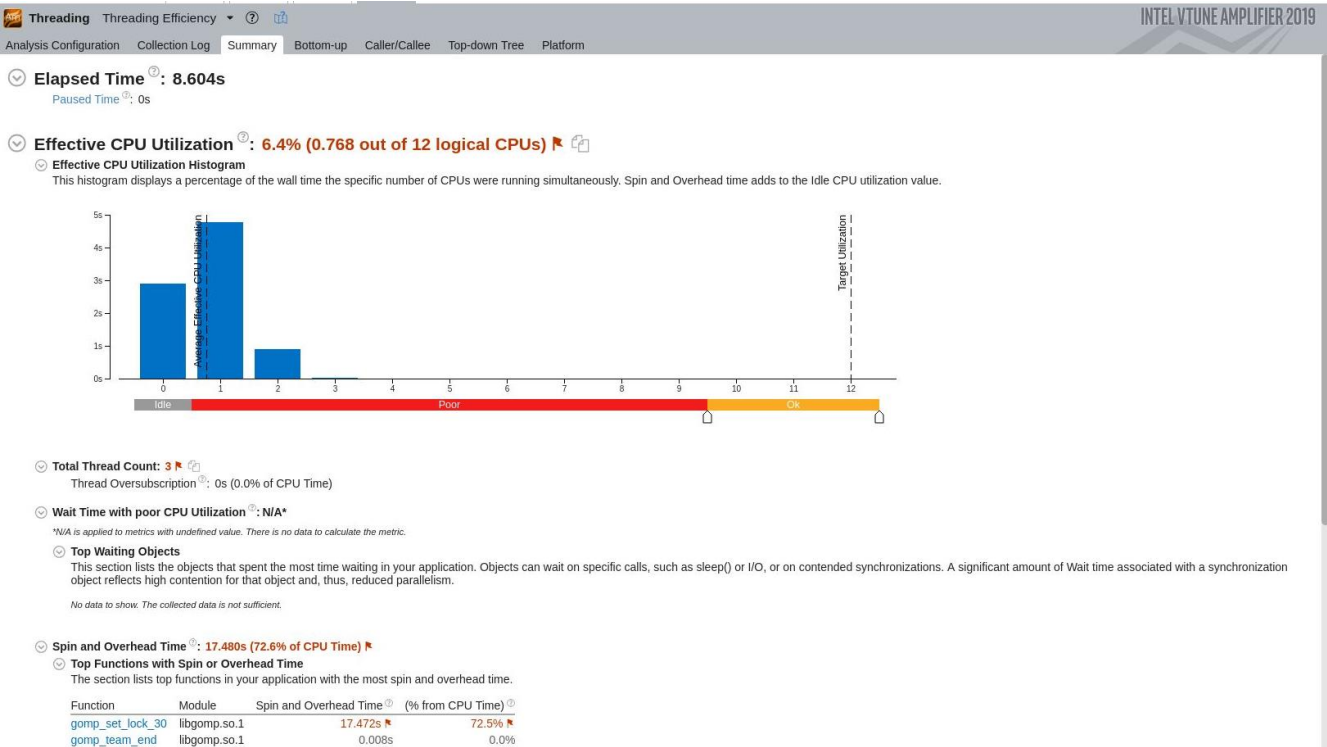


5. Замените строку `#pragma omp critical`. Введите в программу изменения: перед инкрементом переменной `inc` необходимо поставить вызов `omp_set_lock (&writelock)`, после него вызов `omp_unset_lock (&writelock)`. Пример правильного использования этих двух функций показан на изображении `init_lock_openmp.png`.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

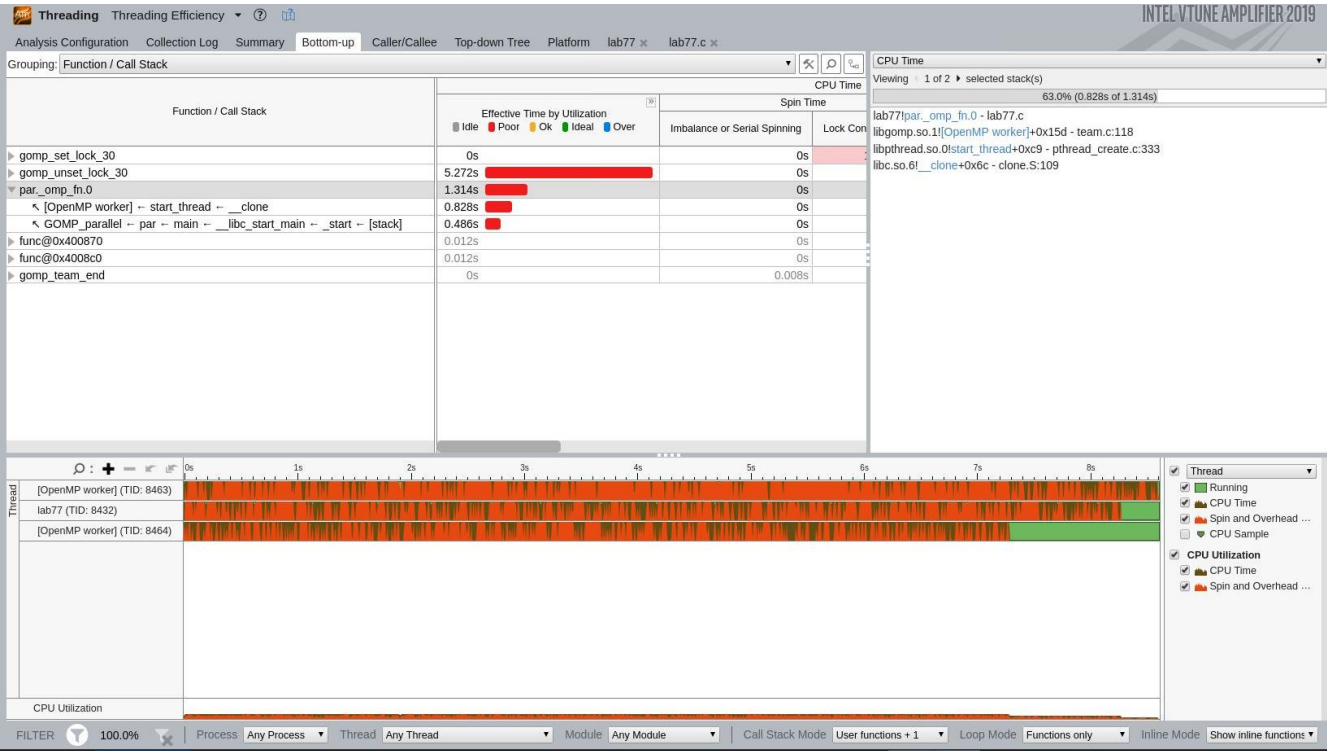
Замена `omp critical` на `writelock`.



Вкладка Summary



Вкладка Bottom-up



Эффективность примерно одинаковая, но у critical больше затрачивается времени на critical_end, а у writelock на set_lock.

Время (в процентах от общего) выполнения строчек кода.

