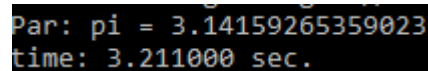


Отчёт по лабораторной работе №5

Петрова О.А. группы ИВТ-12М

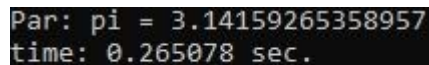
1. Разберите последовательную программу по вычислению определенного интеграла [task_lecture_7.cpp](#). Введите в нее параллелизм с помощью OpenMP. Установите количество рабочих процессов равным 3, для этого используйте оператор `num_threads(num_of_threads)`. Не забудьте настроить в свойствах проекта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



```
Par: pi = 3.14159265359023
time: 3.211000 sec.
```

Рисунок 1. Результат выполнения последовательной программы

По условию задачи требовалось ввести только параллелизм, но т.к. происходила гонка данных было решено добавить редьюсеры.



```
Par: pi = 3.14159265358957
time: 0.265078 sec.
```

Рисунок 2. Результат выполнения изменённой программы

Из результатов видно, что в данной задаче распараллеливание выигрывает во времени по сравнению с последовательным выполнением в 12 раз.

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**.

Elapsed Time[?]: 0.358s

CPU Time[?]: 0.657s


- Effective Time[?]: 0.582s
- Spin Time[?]: 0.075s 
- Overhead Time[?]: 0s

Total Thread Count: 3

Paused Time[?]: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions performance.

Function	Module	CPU Time [?]
par\$omp\$1	IPS_lab5.exe	0.582s
NtYieldExecution	ntdll.dll	0.075s 

**N/A is applied to non-summable metrics.*

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running

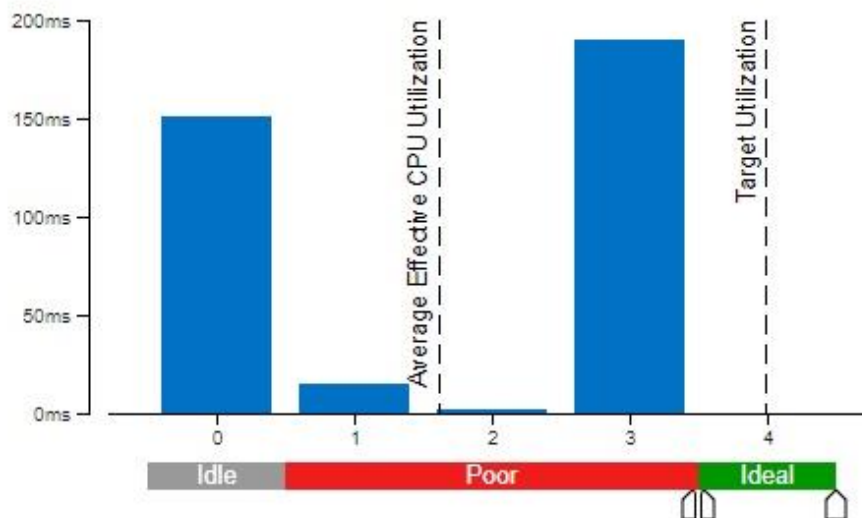


Рисунок 3. Результат работы VTune Amplifier XE

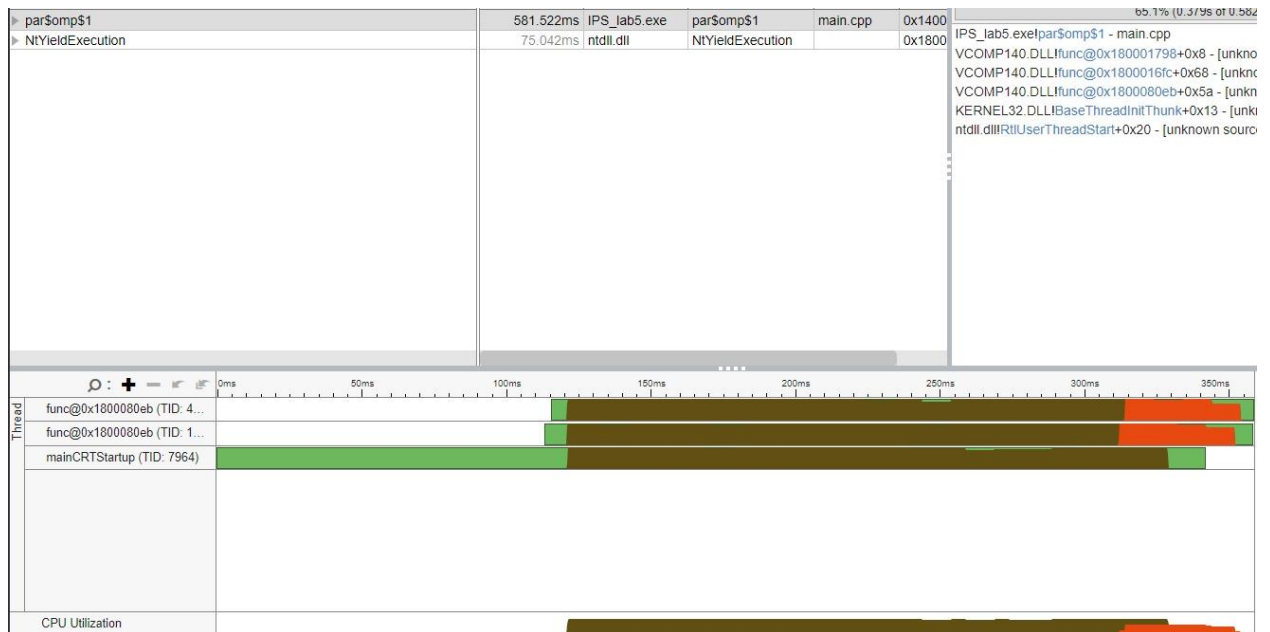


Рисунок 4. Результат работы VTune Amplifier XE во вкладке Bottom-up

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*.

```
Par: pi = 3.14159265358957
time: 1.440865 sec.
```

Рисунок 5. Результат выполнения изменённой программы

Elapsed Time[?]: 2.043s

CPU Time[?]: 5.524s

Effective Time[?]: 0.368s

Spin Time[?]: 0.188s

Overhead Time[?]: 4.969s ⬆

Creation[?]: 0s

Scheduling[?]: 0s

Reduction[?]: 0s

Atomics[?]: 4.969s ⬆

Other[?]: 0s

Total Thread Count: 3

Paused Time[?]: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these performance.

Function	Module	CPU Time [?]
vcomp_atomic_add_i8	VCOMP140.DLL	4.893s ⬆
par\$omp\$1	IPS_lab5.exe	0.368s
NtYieldExecution	ntdll.dll	0.105s
[Import thunk vcomp_atomic_add_i8]	IPS_lab5.exe	0.076s
func@0x18000873b	VCOMP140.DLL	0.066s
[Others]		0.016s

*N/A is applied to non-summable metrics.

Рисунок 6. Результат работы VTune Amplifier XE

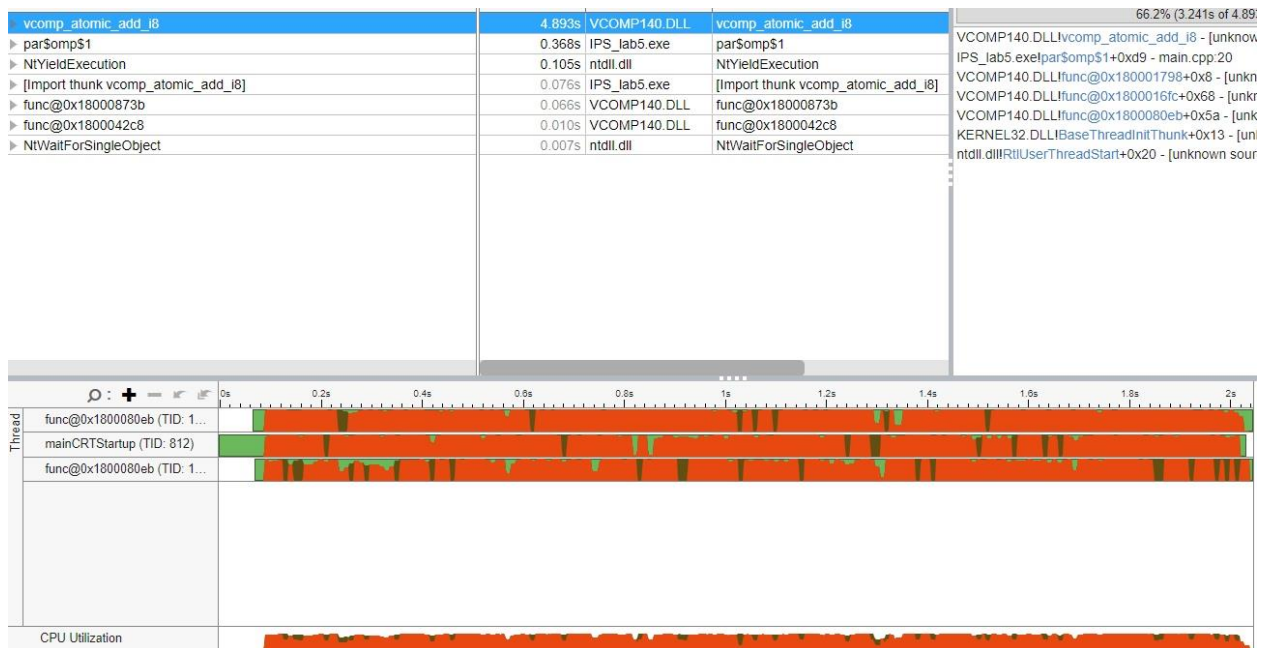


Рисунок 7. Результат работы VTune Amplifier XE во вкладке Bottom-up

Из результатов видно, что большую часть времени было потрачено на ожидание выполнения атомарных операций, поэтому данная программа выполнялась в 5.4 раза дольше предыдущей.

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*.

```
Par: pi = 3.14159265358957
time: 8.401484 sec.
```

Рисунок 8. Результат выполнения изменённой программы

Elapsed Time[?]: 9.605s

CPU Time[?]: 26.778s

Effective Time[?]: 0.884s

Spin Time[?]: 25.894s

Overhead Time[?]: 0s

Total Thread Count: 3

Paused Time[?]: 0s

Top Hotspots

This section lists the most active functions in your application performance.

Function	Module	CPU Time [?]
RtlEnterCriticalSection	ntdll.dll	18.689s
RtlLeaveCriticalSection	ntdll.dll	6.762s
par\$omp\$1	IPS_lab5.exe	0.816s
vcomp_enter_critsect	VCOMP140.DLL	0.249s
NtYieldExecution	ntdll.dll	0.098s
[Others]		0.163s

*N/A is applied to non-summable metrics.

Рисунок 9. Результат работы VTune Amplifier XE

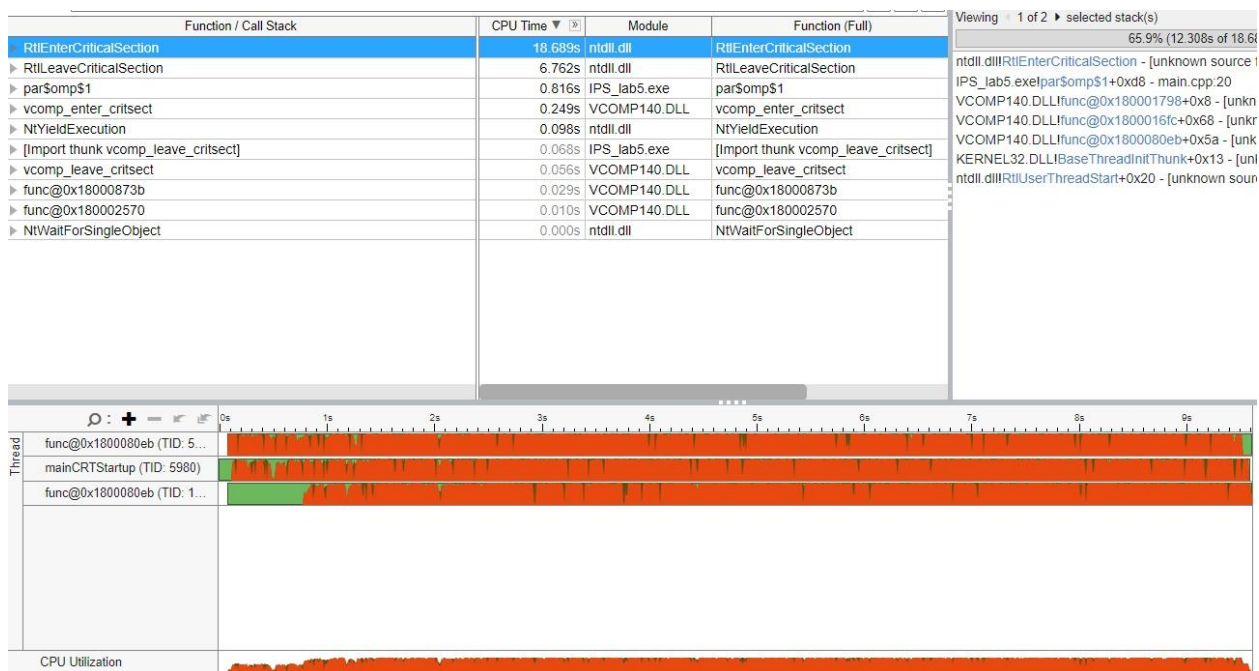
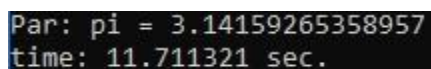


Рисунок 10. Результат работы VTune Amplifier XE во вкладке Bottom-up

Директива `critical` даёт возможность работать с переменной `inc` только одному потоку в определённый момент времени, поэтому время выполнения программы увеличилось.

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.



```
Par: pi = 3.14159265358957
time: 11.711321 sec.
```

Рисунок 11. Результат выполнения изменённой программы

Elapsed Time[?]: 10.931s

CPU Time[?]: 29.412s

Effective Time[?]: 0.639s

Spin Time[?]: 28.773s

Overhead Time[?]: 0s

Total Thread Count: 3

Paused Time[?]: 0s

Top Hotspots

This section lists the most active functions in your application performance.

Function	Module	CPU Time [?]
func@0x1800049d3	VCOMP140.DLL	23.880s
RtlLeaveCriticalSection	ntdll.dll	4.704s
par\$omp\$1	IPS_lab5.exe	0.623s
NtYieldExecution	ntdll.dll	0.164s
exit	ucrtbase.dll	0.016s
[Others]		0.025s

*N/A is applied to non-summable metrics.

Рисунок 12. Результат работы VTune Amplifier XE

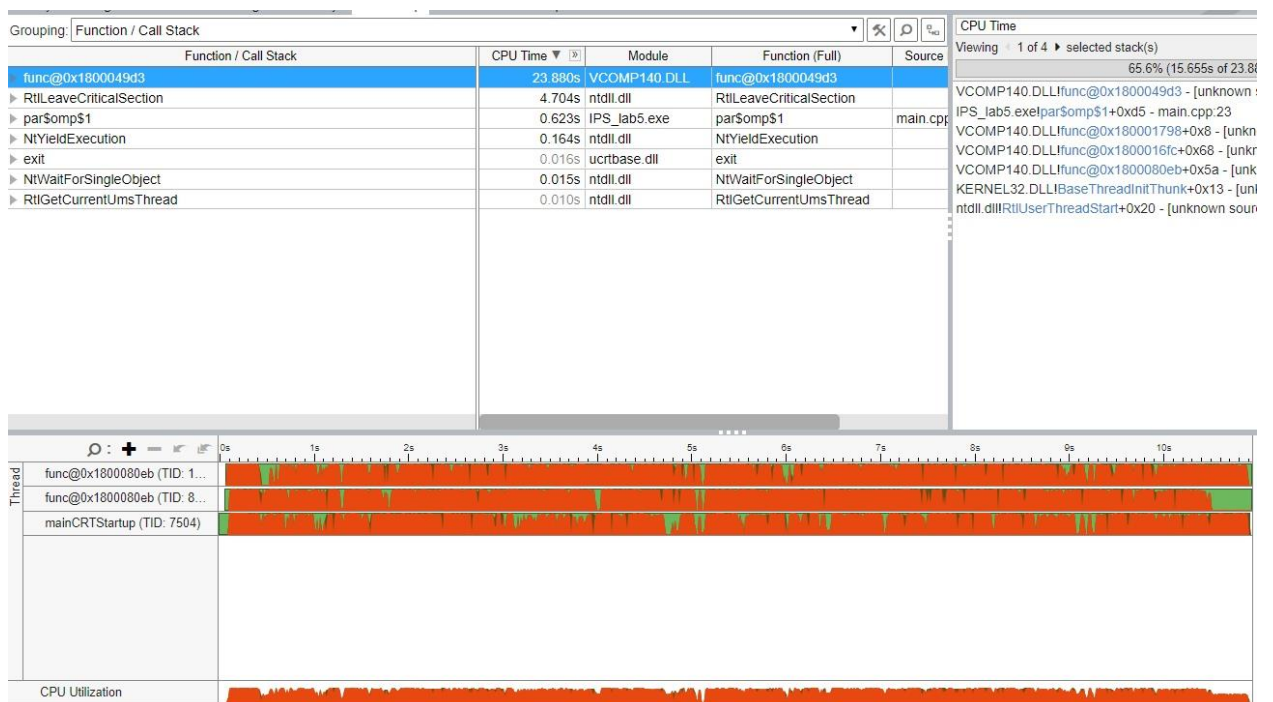


Рисунок 13. Результат работы VTune Amplifier XE во вкладке Bottom-up

Использование замков(lock) подобно использованию директивы critical, но выполнение с ними происходит дольше из-за того, что для начала замок нужно установить, а после выполнения заданных операций снять замок.