План работы

Для данного мастер класса потребуется собирать и запускать 2 приложения:

- mmult (1_mmult) Задача по нахождению результата выполнения операции матричного умножения для двух матриц размерности N*N, где N - аргумент, который можно задать при запуске приложения, по умолчанию - 1024.
- primary_nums (6_primary_nums) Задача по нахождению всех простых чисел из диапазона 2:N, где N верхняя граница рассматриваемого множества натуральных чисел.

В рамках данного мастер-класса необходимо будет:

1. Изучить прилагаемый документ по основам использования программной модели OMP, способы создания параллельных регионов, распределения работы между потоками и методы по защите разделяемых данных.
2. Использовать OMP директивы препроцессора для создания параллельной версии mmult.
3. Проанализировать эффективность параллельной версии перемножения матриц с использованием инструмента Intel VTune, сделать вывод по результатам анализа
4. Проанализировать эффективность параллельной версии приложения по нахождению множества всех простых чисел на заданном диапазоне
5. Применить модификации простейшего решения в случае, если удается обнаружить проблему в производительности
6. Сравнить две реализации, сделать выводы

- Настройка окружения
- Введение в OpenMP

Рекомендации по выполнению работы

Для успешного выполнения работы необходимо ознакомиться с документами выше. Добавление параллельных регионов возможно только в тех местах (например, в циклах, где каждая итерация будет исполняться в параллель т.е. будет параллельной задачей), где нет зависимостей между параллельными задачами. Определить это поможет уже изученный на предыдущих занятиях инструмент - Intel Advisor и его Dependencies Analys

Потенциальный прирост производительности за счет распараллеивания можно узнать при помощи запуска Suitability анализа, который выполняли на МК2.

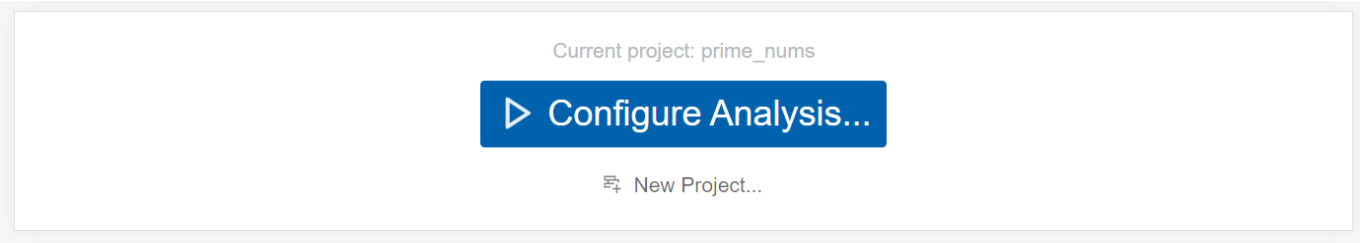
Последовательность запусков:

- Проверка корректности работы программы: запуск небольшой по размеру задачи, сравнение с эталонным значением
- Запуск последовательной версии на большой задаче (большее значение входного аргумента)
- Добавление параллелизма, сравнение времени выполнения и результатов работы с последовательной версией
- Анализ параллельного приложения в VTune, оценка эффективности.

Анализ в VTune

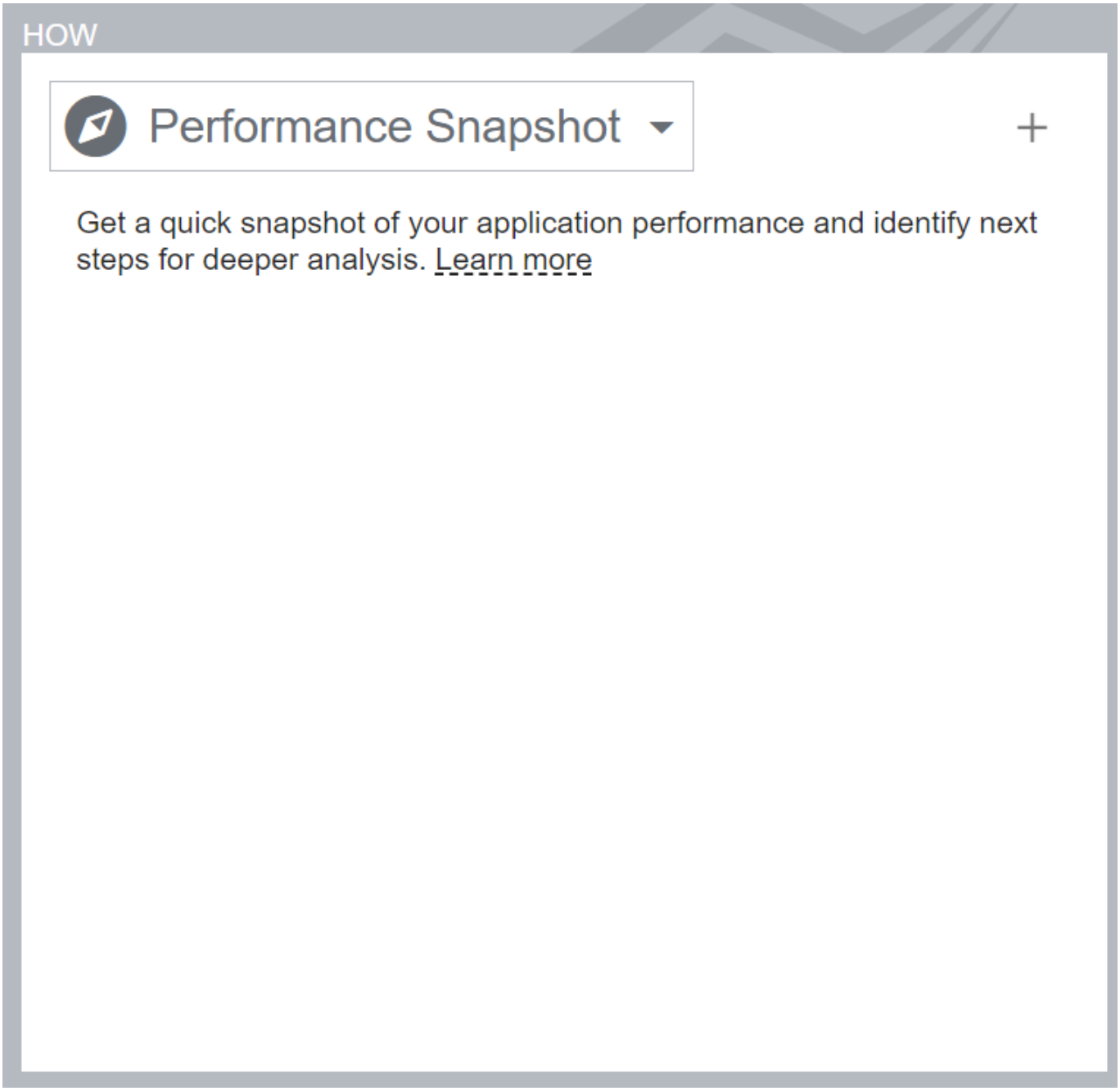
Mmult

Для того чтобы выполнить анализ производительности параллельного OMP приложения в Intel VTune, необходимо создать в нем новый проект (New Project)

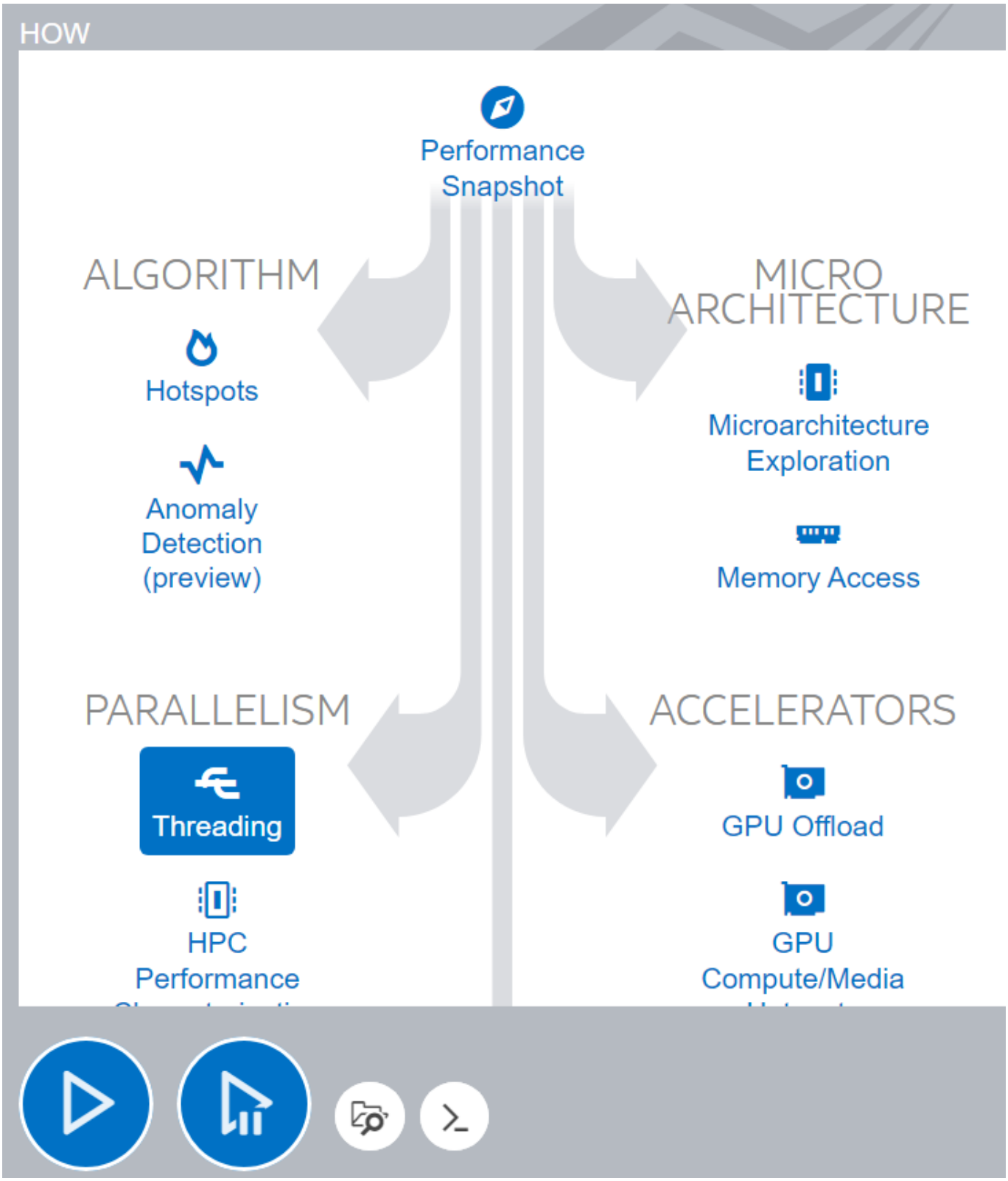


Далее все стандартно: выбираем место для сохранения проекта, название проекта, анализируемое приложение и аргументы для него (mmult с >= 4000 размерностью матриц)

Далее в **Performance Snapshot**



Необходимо выбрать **Threading** анализ



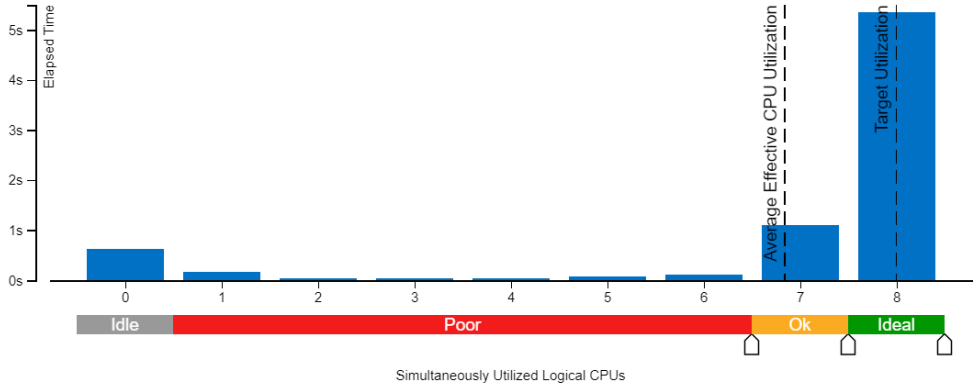
И запустить его.

В собранном результате первым делом необходимо обратить внимание на **Summary** страницу.

Effective CPU Utilization: 85.6% (6.844 out of 8 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



На ней представлена информация об эффективной утилизации CPU. Это столбчатая диаграмма, на которой сумма длин всех столбцов будет показывать **Elapsed time** - это время от старта программы до ее завершения, то есть, без учета того, сколько каждый поток времени отработал по отдельности.

Если же взять в учет кол-во потоков для каждого столбца, и домножить эти значения на временные велечины для этих столбцов, то мы получим уже **Total time**.

Собственно, смысл этого графика показать, какую долю от всего времени программы (от Elapsed time) мы в полную меру утилизировали все доступные физические потоки.

Далее приведена статистика непосредственно по OMP модели

OpenMP Analysis. Collection Time: 7.500

Serial Time (outside parallel regions): 0.761s (10.1%)

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the Wait time of the master thread, it may significantly exceed the aggregated CPU time in the table.

Function	Module	Serial CPU Time
transpose<float>	mmult.exe	0.102s
free_dbg	ucrtbased.dll	0.077s
exit	ucrtbased.dll	0.010s

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

Parallel Region Time: 6.740s (89.9%)

Estimated Ideal Time: 6.675s (89.0%)

OpenMP Potential Gain: 0.065s (0.9%)

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
??$multiply_simple@M@@YAXHPEAPEAM00@Z.extractedompparallel:8@C:\Users\konstantin\Documents\workfolder\EHC-LABS/1_mmult\mmult.cpp:53:53	0.058s	0.8%	6.740s

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

Поскольку собирали мы приложение с компилятором от Intel, он при генерации OMP кода вставил свою отладочную информацию, по которой в дальнейшем профилировщик смог определить, когда происходит процесс создания параллельного региона, а когда выполняется последовательный участок программы. Это **parallel** и **serial** блоки.

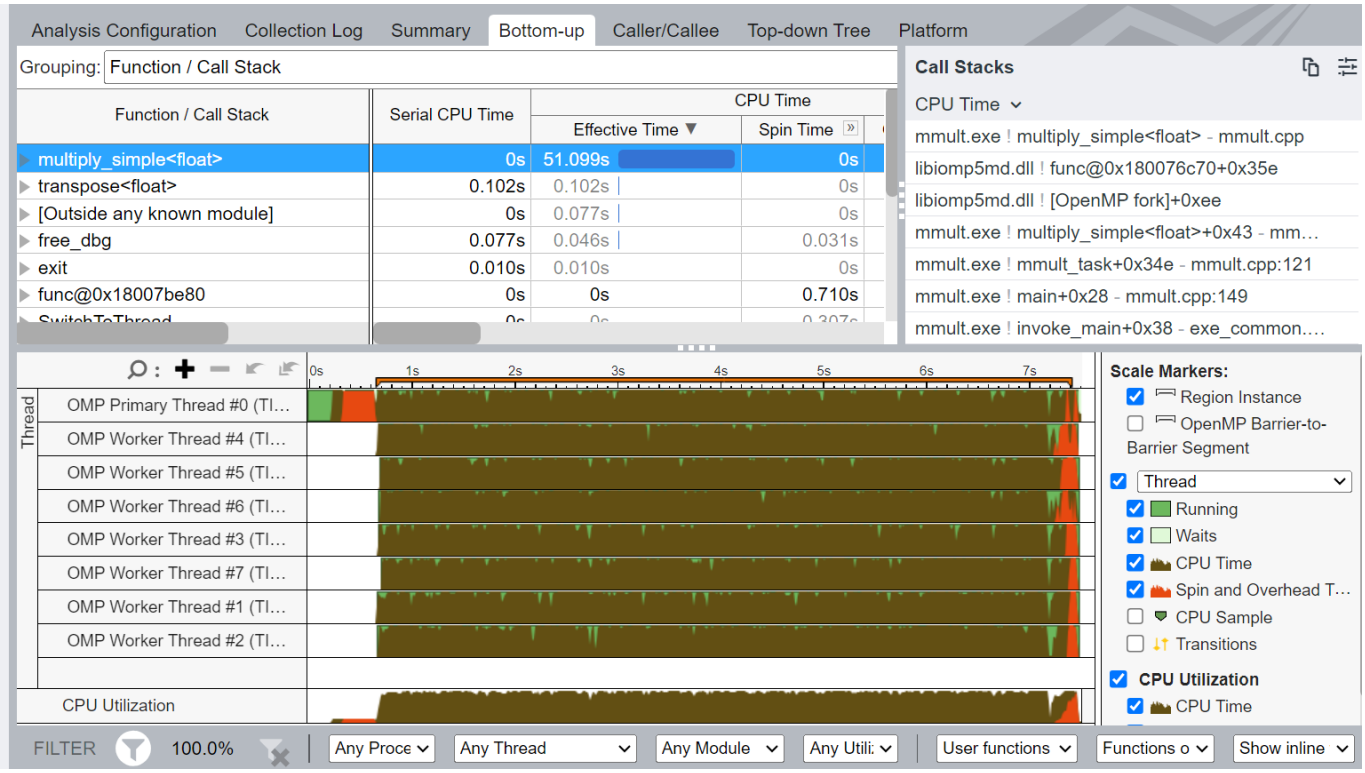
Далее следует общая статистика утилизации CPU, сколько времени он крутился в активном ожидании (spin and owerhead), либо же ожидал каких-то событий, но не занимал процессорное время. Это все так же доступно и для OMP специфичных задач по созданию потоков, их синхронизации.

Summary

Для данного примера MMULT приложения (отмечу, на большом размере задачи, когда доля последовательной части уже не так велика), мы имеем очень хороший случай отлично распараллеливаемой задачи. Так же стоит отметить, что при росте сложности самой задачи, доля параллельного региона только растет, и он легко занимает всю доступную ширину вычислительных возможностей (все доступные ядра). Пример хорошо масштабируемой задачи.

Дополнительное задание: посмотреть на соотношение serial/parallel частей анализируемого приложения на разных размерах задачи по перемножению матриц (разная размерность самих матриц, например от 10 до 1000).

На **Bottom-up** вкладке можно обратить внимание на временной график утилизации CPU, там тоже все замечательно:



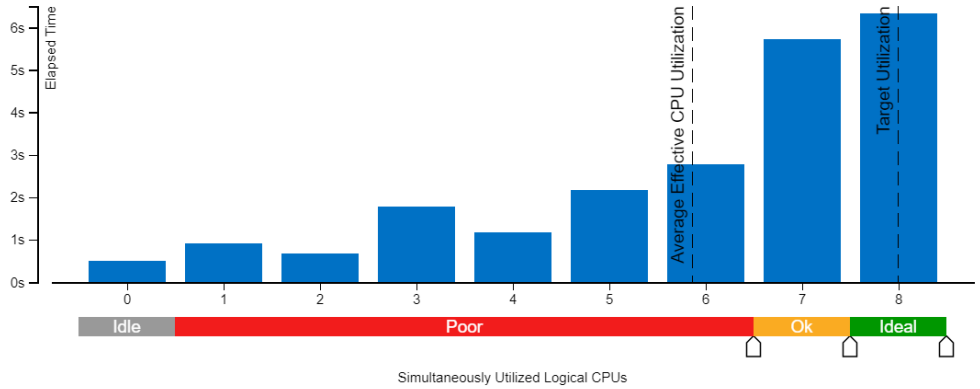
Prime numbers

В данном примере уже получается картина интереснее

Effective CPU Utilization: 73.3% (5.868 out of 8 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



Поскольку работа по итерациям распределена не равномерно (последние для большего N работают дольше), то и при статическом распределении задач по процессорам мы получим простои на тех ядрах, на которые назначали первые итерации.

Тут же возрастет Wait time, как раз из-за того, что менее загруженные потоки будут ожидать завершения остальных потоков:

Wait Time with poor CPU Utilization: 33.623s (88.9% of Wait Time)

Top Waiting Objects

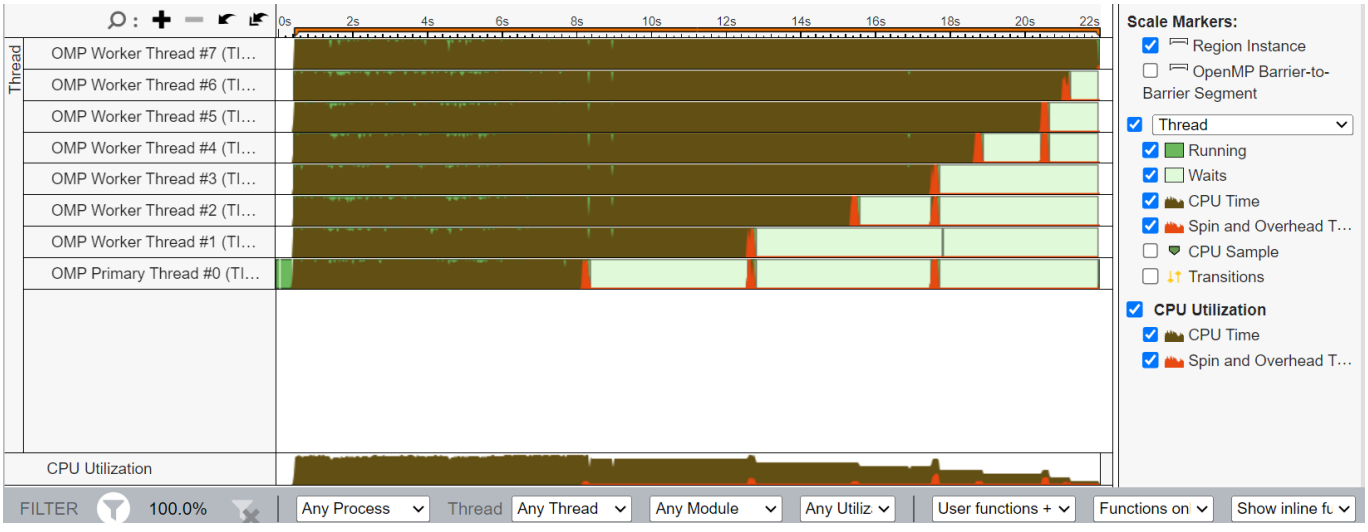
This section lists the objects that spent the most time waiting in your application. Objects can wait on specific calls, such as sleep() or I/O, or on contended synchronizations. A significant amount of Wait time associated with a synchronization object reflects high contention for that object and, thus, reduced parallelism.

Sync Object	Wait Time with poor CPU Utilization	(% from Object Wait Time)	Wait Count
Manual Reset Event 0xaa81432a	18.854s	81.9%	8
Manual Reset Event 0xd2e83a15	13.431s	100.0%	3
Manual Reset Event 0xbcd364aa	1.330s	100.0%	2
Multiple Objects	0.007s	100.0%	1
Stream 0xcb838727	0.000s	100.0%	1
[Others]	0.000s	100.0%	2

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

Spin and Overhead Time: 2.121s (1.6% of CPU Time)

Картина в bottom-up говорит то же самое



Поэтому тут в OMP директиве нужно указать, чтобы он попытался распределять задачи динамически:

```
#pragma omp parallel for schedule(dynamic) reduction(+:sum)
```

И вот тогда уже процент эффективности использования многопоточной системы будет гораздо выше

Effective CPU Utilization: 93.6% (7.486 out of 8 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.