

# Intel Thread Profiler

## Краткое описание

### Пример использования Intel® Thread Profiler

Целью настоящего документа является начальное ознакомление с инструментом ИТР и общими принципами работы с ним. Изучается процесс подготовки приложения к профилированию, графический интерфейс ИТР и основные возможности по анализу производительности многопоточных приложений.

#### Изучение профилируемого приложения

Лабораторная работа проводится на учебном приложении, которое осуществляет факторизацию (разложение на простые множители) чисел из диапазона от 1 до N. Используется алгоритм, который основан на попытке деления факторизируемого числа на каждое из меньших его чисел. Если остаток от деления равен нулю, то очередной множитель запоминается, после чего производится повторная попытка деления на это же число. При нахождении каждого множителя, факторизируемое число делится на него, и алгоритм завершает работу, когда частное от очередного деления становится равным единице.

$12 = ? * ? * \dots * ?$

$12 / 2 = 6$ ; // пробуем разделить на 2 еще раз

$6 / 2 = 3$ ; // пробуем разделить на 2 еще раз

$3 / 2 = 1,5$ ; // берем следующий делитель

$3 / 3 = 1$ ; // СТОП! – получили единицу

Заметим, что это малоэффективный алгоритм факторизации, поэтому мы не рекомендуем использовать его при решении практических задач. Использование такого алгоритма предпринято только в учебных целях - на его примере мы сможем изучить некоторые особенности оптимизации многопоточных приложений.

Откройте проект **Factorization**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**,
- в меню **File** выполните команду **Open→Project/Solution...**,
- в диалоговом окне **Open Project** выберите папку **D:\Factorization**,
- дважды щелкните на файле **Factorization.sln** или, выбрав файл, выполните команду **Open**.

В окне **Solution Explorer** дважды щелкните на файле исходного кода **Factorization.cpp** (Рисунок 1). После этого в рабочей области **Microsoft Visual Studio** появится программный код, с которым нам предстоит работать.

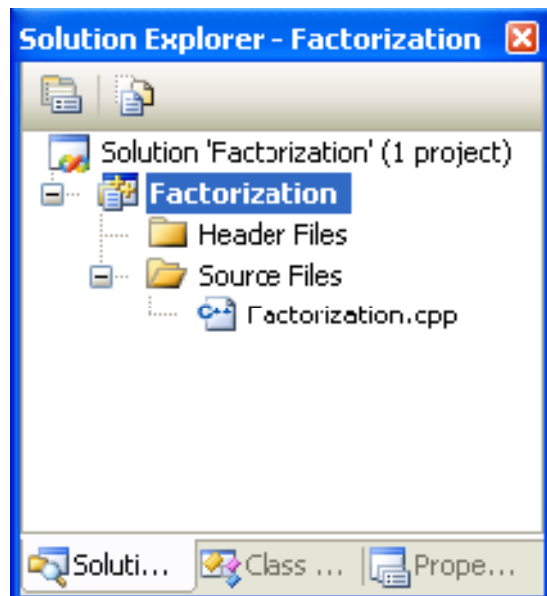


Рисунок 1 - Открытие файла Factorization.cpp

Приступим к изучению приложения. В начале файла **Factorization.cpp** объявлены две константы:

```
#define NUM_NUMBERS 100000  
#define NUM_THREADS 2
```

Первая из них указывает количество чисел, которые будут факторизованы. В данном случае будет построено разложение для чисел от 1 до 100000. Вторая константа показывает, сколько потоков будет создано для решения этой задачи.

Также объявлен глобальный массив векторов **divisors**.

```
vector<int> divisors[NUM_NUMBERS+1];
```

Он предназначен для хранения простых множителей каждого из чисел. Так, например, вектор **divisors** для NUM\_NUMBERS=6 будет содержать два числа: 2 и 3.

```
divisors[5] = {5}  
divisors[6] = {2, 3}
```

В данной лабораторной работе нас будут интересовать только функции **main** и **factorization1**.

Ознакомьтесь с кодом функции **main**. Он содержит объявления переменных, операции создания потоков и ожидания их завершения, а также вывод на экран, предназначенный для контроля правильности результатов.

После этого ознакомьтесь с функцией **factorization1**, которая представляет собой рабочую функцию потока. В ней реализуется простейшая стратегия распределения нагрузки между потоками. А именно, первый поток строит разложение для первых

NUM\_NUMBERS/NUM\_THREADS чисел, второй – для второго массива чисел такой же длины, и так далее. Пример распределения нагрузки между двумя потоками представлен на рисунке 2.

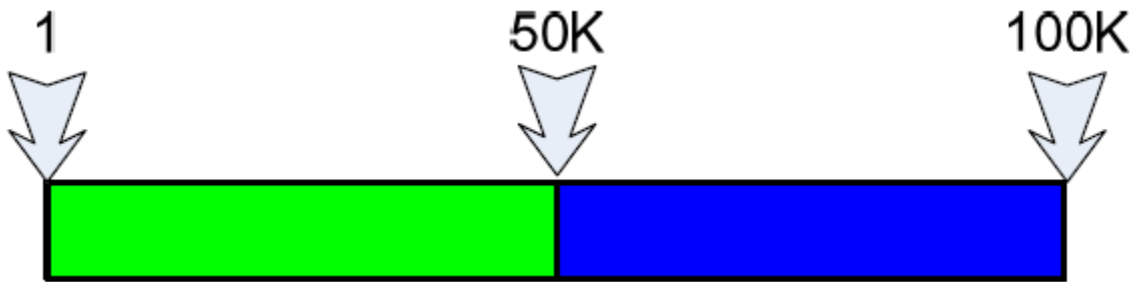


Рисунок 2 - Распределение нагрузки между потоками

Скомпилируйте и запустите приложение:

- В меню **Build** выберите команду **Build Solution**;
- В меню **Debug** выберите команду **Start Without Debugging**.

Убедитесь в правильности работы приложения по выводу на консоли.

### Подготовка приложения для профилирования

Для того чтобы приложение можно было профилировать при помощи ИТР, необходимо установить определенные настройки компиляции и компоновки проекта. Большинство из них установлены по умолчанию, но некоторые необходимо указывать самостоятельно.

1. В меню **Build** выберите пункт **Configuration Manager...**, в открытом окне выберите режим **Release**.

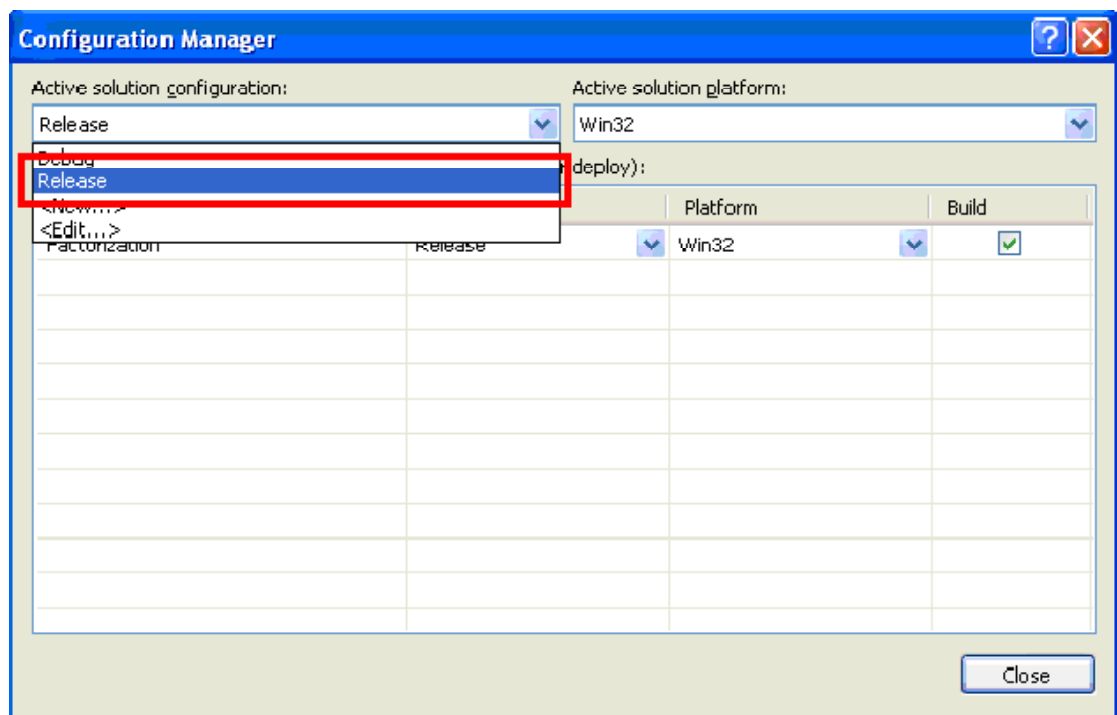


Рисунок 3 - Выбор режима Release

2. В меню **Project** выберите пункт **Properties**, в результате чего появится окно, представленное на рисунке 4. В дереве слева выберите узел **Configuration Properties→C/C++→General**. В открывшейся таблице справа для элемента с именем **Debug Information Format**, установите значение **Program Database (/Zi)**.

Нажмите кнопку **Apply**.

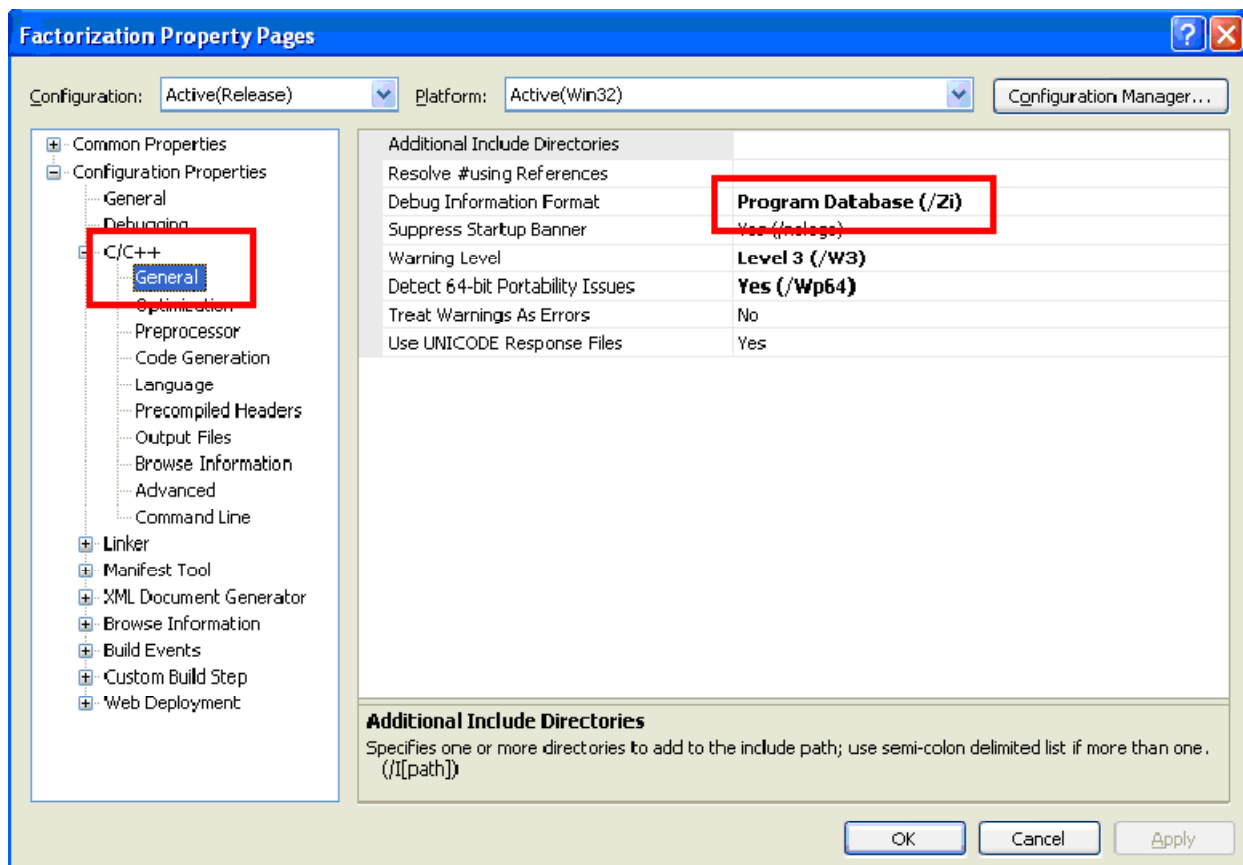


Рисунок 4 - Указание формата отладочной информации

3. В этом же окне настроек проекта необходимо убедиться в том, что для приложения генерируется отладочная информация. В дереве слева выберите узел **Configuration Properties→Linker→Debugging**. В открывшейся таблице справа для элемента с именем **Generate Debug Info**, необходимо установить значение **Yes (/DEBUG)**.

После этого нажмите кнопку **Apply**.

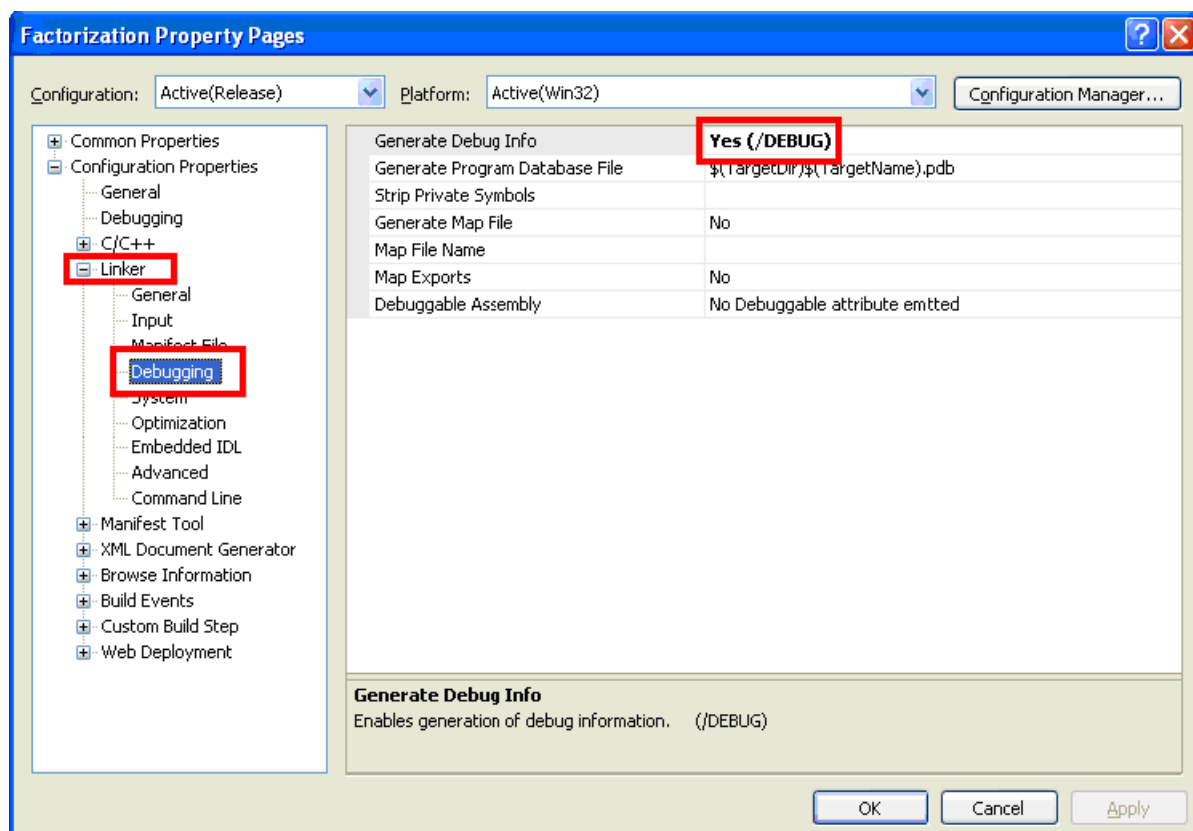


Рисунок 5 - Указание генерации отладочной информации

4. Убедитесь, что используются потокобезопасные библиотеки. Для этого выберите узел **Configuration Properties**→**C/C++**→**Code Generation**. В открывшейся таблице справа для элемента с именем **Runtime library** установите значение **Multi-threaded DLL (/MD)**. Нажмите кнопку **Apply**.

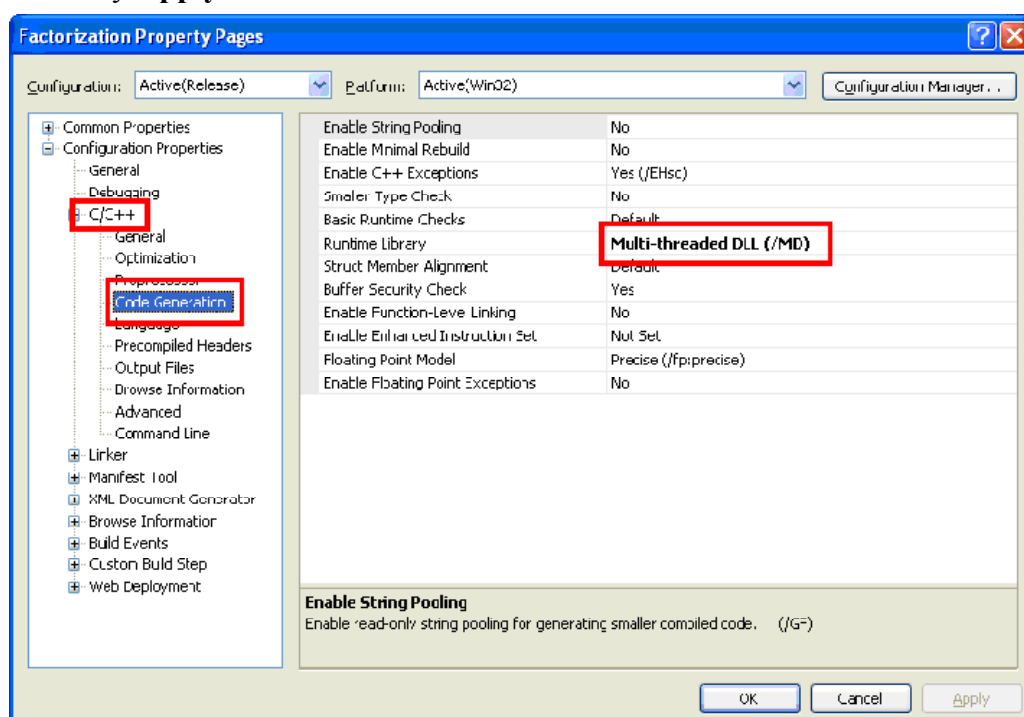


Рисунок 6 - Выбор потокобезопасных библиотек

5. Убедитесь, что приложение компонуется с использованием опции **/fixed:no**. В окне настроек проекта выберите узел **Configuration Properties→Linker→Advanced**. В открывшейся таблице справа для элемента с именем **Fixed Base Address** установите значение **Generate a relocation section (/FIXED:NO)**.

Нажмите кнопку **Apply**.

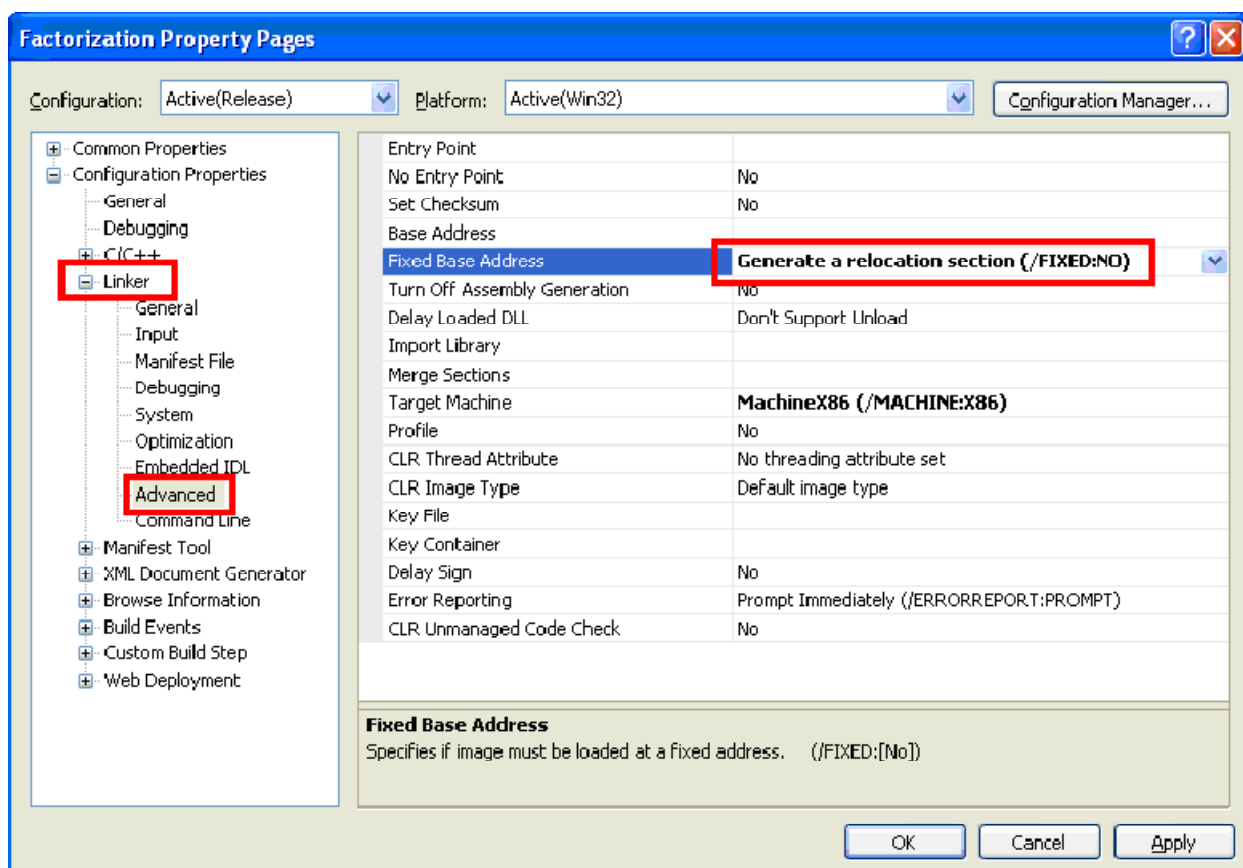


Рисунок 7 - Установка опций компоновщика

После выполнения данных процедур Ваше приложение готово к профилированию.

## Профилирование приложения

### Создание проекта Intel® Thread Profiler

- 1 Запустите Intel® Thread Profiler. Найти его можно, например, по следующему пути: **Start→All programs→Intel(R) Software Development Tools→Intel(R) Thread Profiler 3.1→Intel(R) Thread Profiler**.
- 2 В открывшемся окне нажмите на кнопку **New Project**.
- 3 В новом окне выберите **Intel(R) Thread Profiler Wizard** и нажмите кнопку **OK**.

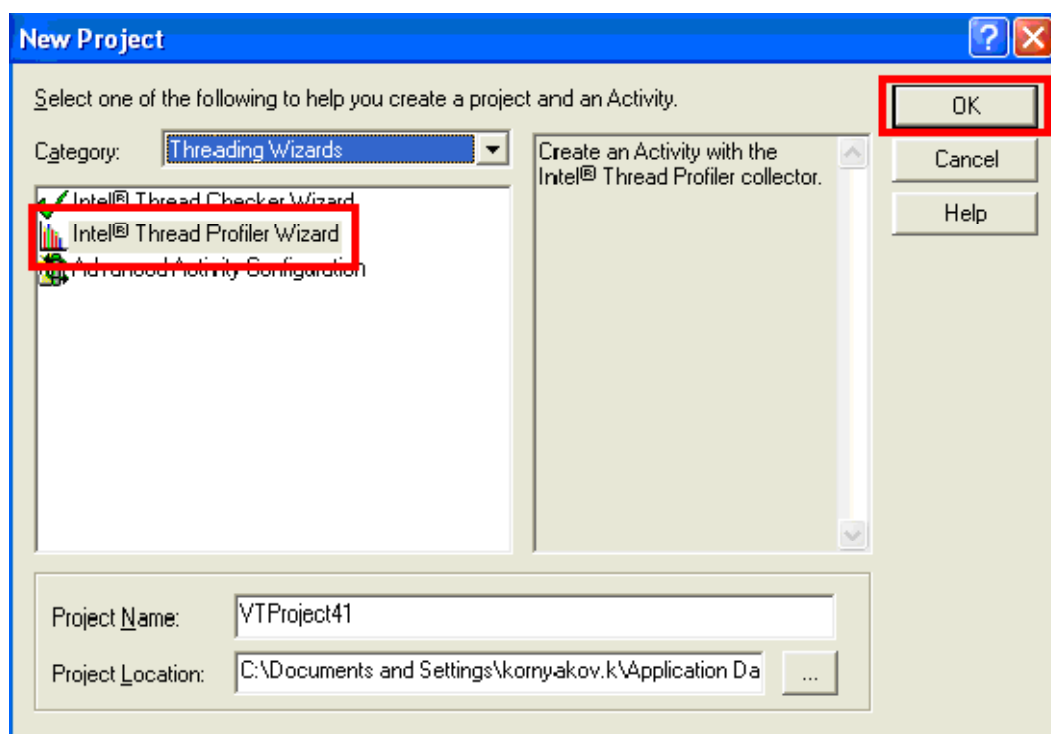


Рисунок 8 - Выбор типа проекта

4 Ниже надписи **Launch an application** имеется строка, в которой необходимо указать имя профилируемого приложения. Нажмите кнопку [...] и в открывшемся диалоговом окне укажите путь до приложения, подлежащего профилированию. В нашем случае это **D:\Factorization\Release\Factorization.exe**.

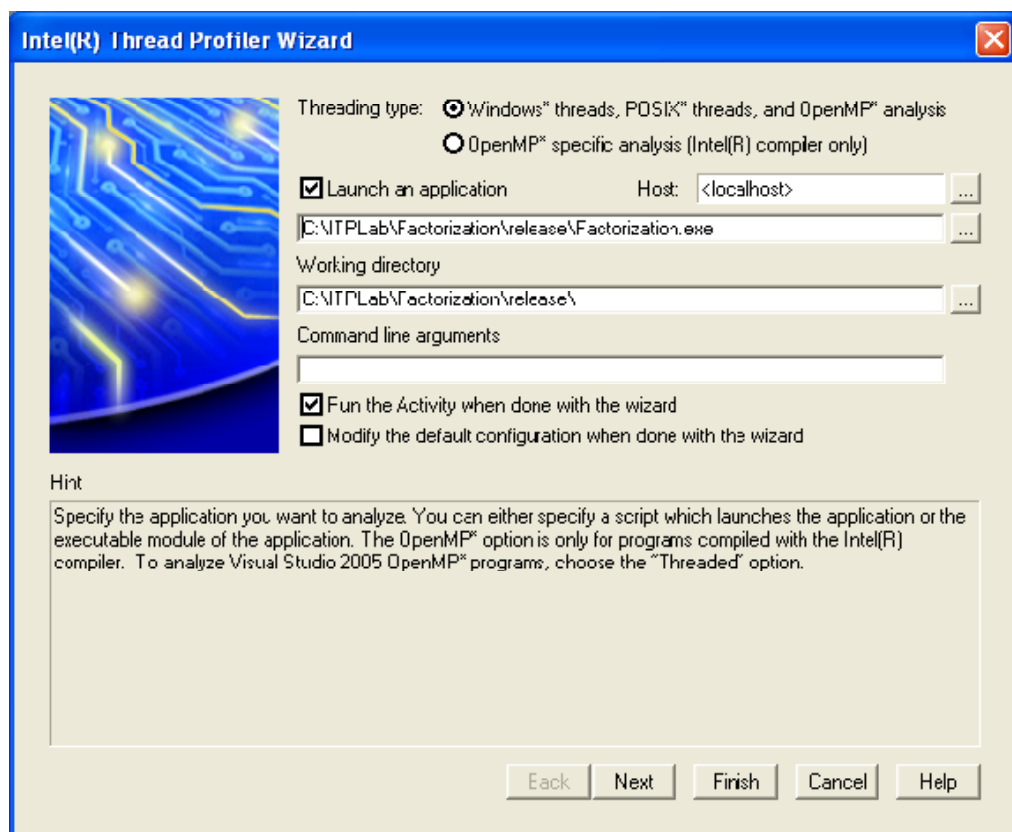


Рисунок 9 - Выбор профилируемого приложения

5 Нажмите кнопку **Finish**.

После этого ITP произведет инструментацию вашего приложения и немедленно начнет профилирование. Когда сбор трассы завершится, информация о ней появится на экране. Окно ITP содержит следующие элементы: панель инструментов, браузер результатов запусков (Tuning Browser), окно сообщений Output и окна Profile и Timeline (Рисунок 10). Последние два окна являются основными источниками информации о критическом пути приложения.

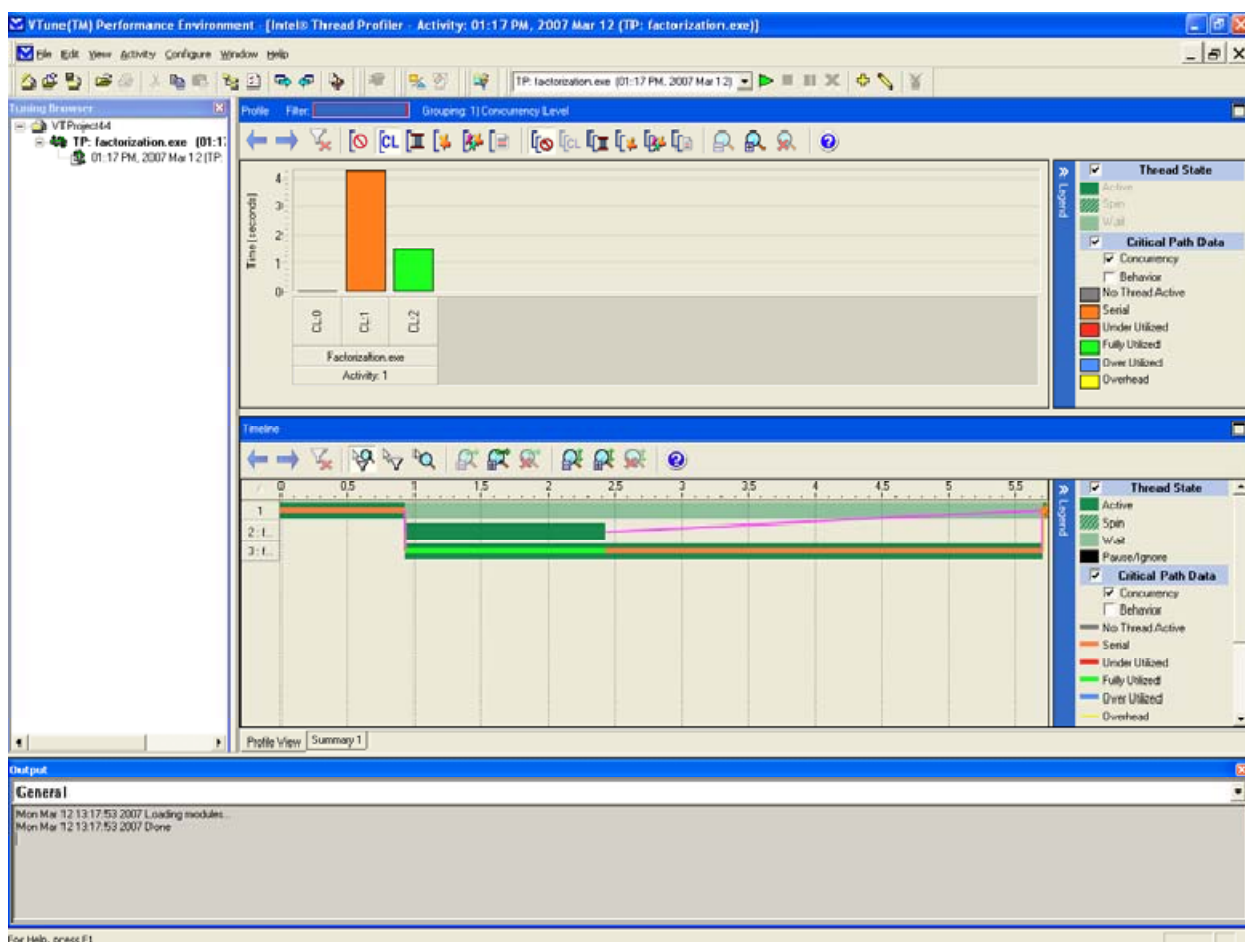


Рисунок 10 - Рабочая область Intel® Thread Profiler

## Профилирование

Профилирование приложения можно начать несколькими способами:

- Выбрать пункт меню **Activity**→**Run**,
- Нажать **F5**,
- Нажать на панели инструментов кнопку с зеленой стрелкой.

Результаты каждого запуска сохраняются, и вы можете работать с несколькими профилями одновременно. Доступ к профилям осуществляется через Tuning Browser (View→Tuning Browser).

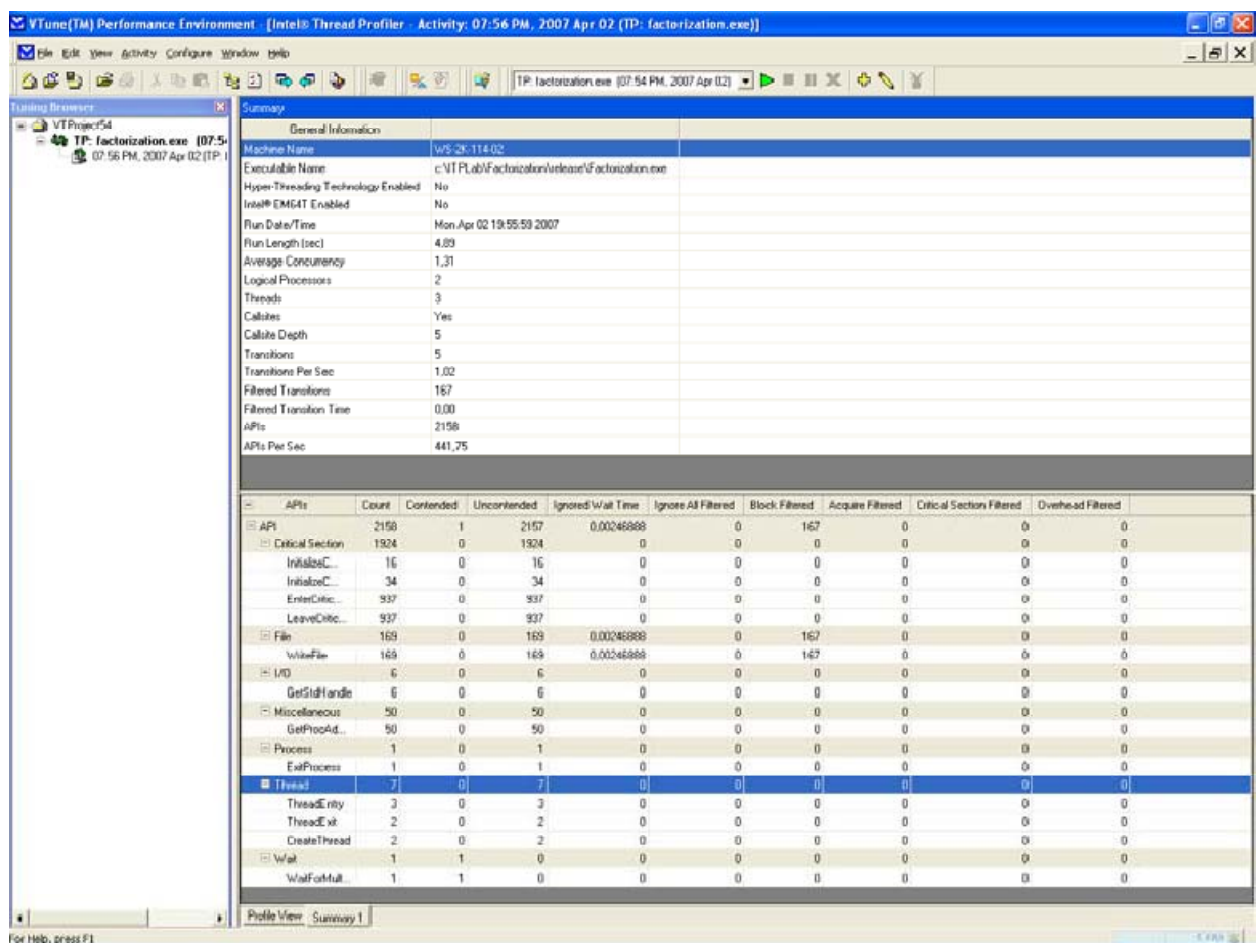


## Анализ производительности приложения

Далее мы по шагам изучим основные элементы графического интерфейса ИТР, попутно рассматривая, как каждый из них используется при анализе производительности приложения.

### Вкладка Summary

В нижней части окна ИТР располагаются вкладки под названием Profile View и Summary. Перейдите на вторую из них, наведя на нее курсор и щелкнув левой кнопкой мыши. Эта вкладка содержит общую информацию о трассе приложения (Рисунок 11).



The screenshot shows the 'Summary' tab in the VTune Performance Environment. It displays general information about the application 'factorization.exe' and a detailed table of API calls.

General Information										
Machine Name	WS-2K-114-02									
Executable Name	c:\VT\Lab\Factorization\release\Factorization.exe									
Hyper-Threading Technology Enabled	No									
Intel® EM64T Enabled	No									
Run Date/Time	Mon Apr 02 19:55:59 2007									
Run Length (sec)	4.89									
Average Concurrency	1.31									
Logical Processors	2									
Threads	3									
Calikes	Yes									
Calike Depth	5									
Transitions	5									
Transitions Per Sec	1.02									
Filtered Transitions	167									
Filtered Transition Time	0.00									
APIs	2156									
APIs Per Sec	441.25									
APIs	Count	Contented	Uncontented	Ignored/Wait Time	Ignore All Filtered	Block Filtered	Acquire Filtered	Critical Section Filtered	Overhead Filtered	
API	2156	1	2157	0.00246888	0	167	0	0	0	0
Critical Section	1924	0	1924	0	0	0	0	0	0	0
InitializeC...	16	0	16	0	0	0	0	0	0	0
InitializeC...	34	0	34	0	0	0	0	0	0	0
EnterCriti...	937	0	937	0	0	0	0	0	0	0
LeaveCriti...	937	0	937	0	0	0	0	0	0	0
File	169	0	169	0.00246888	0	167	0	0	0	0
WriteFile	169	0	169	0.00246888	0	167	0	0	0	0
I/O	6	0	6	0	0	0	0	0	0	0
GetStdHandle	6	0	6	0	0	0	0	0	0	0
Miscellaneous	50	0	50	0	0	0	0	0	0	0
GetProcAd...	50	0	50	0	0	0	0	0	0	0
Process	1	0	1	0	0	0	0	0	0	0
ExitProcess	1	0	1	0	0	0	0	0	0	0
Thread	2	0	2	0	0	0	0	0	0	0
ThreadExi...	3	0	3	0	0	0	0	0	0	0
ThreadYit...	2	0	2	0	0	0	0	0	0	0
CreateThread	2	0	2	0	0	0	0	0	0	0
Wait	1	1	0	0	0	0	0	0	0	0
WaitForMul...	1	1	0	0	0	0	0	0	0	0

Рисунок 11 - Вкладка Summary

Вкладка состоит из двух таблиц: первая содержит общую информацию о состоявшемся запуске (характеристики узла, время работы приложения, число потоков и т.д.), а вторая – статистику вызовов API.

Щелкните на плюс, располагающийся возле надписи “APIs” в нижней таблице, при этом она должна принять вид, как показано на рисунке 11. Из таблицы можно узнать, сколько времени приложение потратило на ввод/вывод, синхронизацию, непроизводительные издержки и т.д.

После того как вы ознакомитесь с приведенной информацией, вернитесь на вкладку Profile View.

## Окно Profile

Основную часть окна Profile занимает область, на которой представлен критический путь приложения.

Пользователь имеет возможность управлять представлением критического пути, для чего используется функция группировки и легенда, располагающаяся справа. Рассмотрим их подробнее.

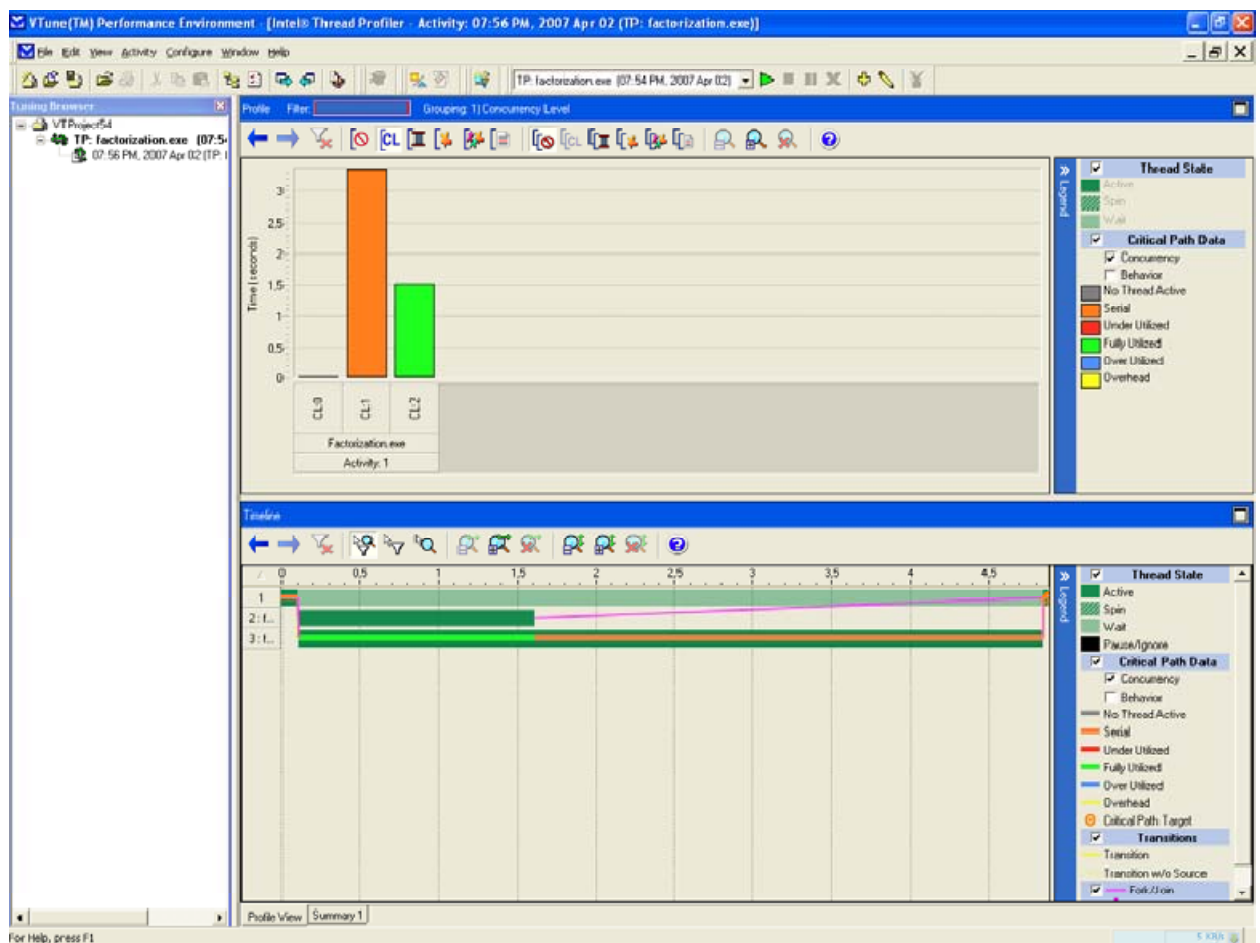


Рисунок 12 - Окно Profile

## Выбор характеристик поведения потоков для визуализации

С помощью панели Legend пользователь имеет возможность указать интересующие его характеристики поведения потоков. Для этого используются находящиеся на легенде кнопки-флажки (check-box). Первый из них, называемый **Thread State**, позволяет указать, интересует ли нас информация о состоянии потока.

Напомним, что существуют три типа состояния: активное (active), активное ожидание (spin), ожидание (wait).

Выберите тип группировки по потокам, нажав на панели кнопку с изображением катушки. После этого в окне появится информация о распределении времени в критическом пути, как показано на рисунке 13.

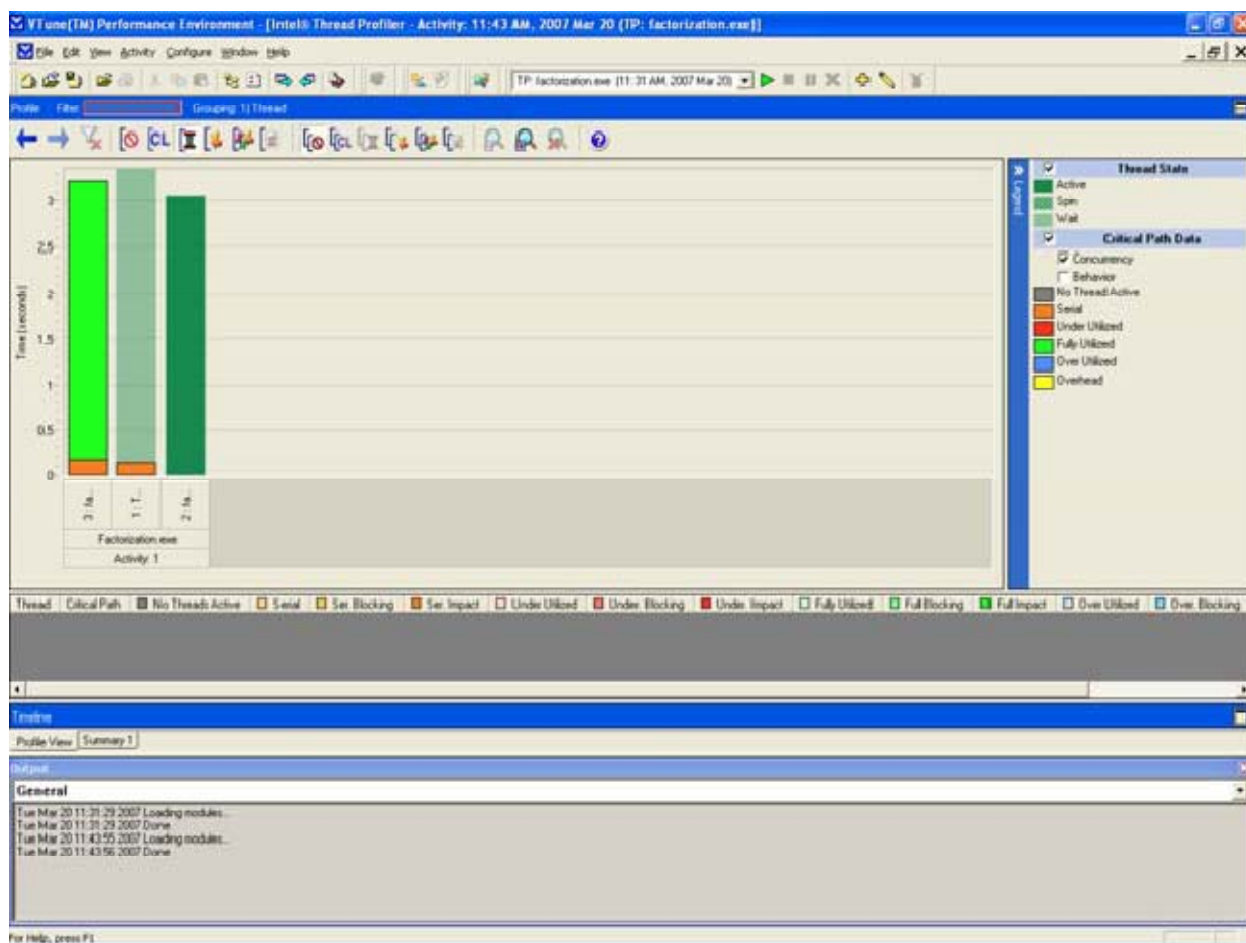


Рисунок 13 - Анализ состояний потоков

Информация о состоянии потоков представлена в виде столбиков зеленого цвета различной насыщенности. Бледно-зеленый цвет используется для обозначения того, что поток находился в состоянии ожидания, а темно-зеленый цвет соответствует активному состоянию потока. Из рисунка можно узнать, что два потока в нашем приложении (первый и третий столбики) большую часть времени были активны, а еще один поток в основном находился в состоянии ожидания. Нетрудно понять, что в ожидании находился основной поток нашего приложения, в то время как созданные им потоки производили разложение чисел на простые множители. Снимите флажок **Thread State** и убедитесь, что информация о состоянии потоков исчезнет. После этого верните флажок в исходное положение.

Следующая кнопка-флажок – **Critical Path Data**, при помощи которой пользователь может указать, интересует ли его разбиение критического пути по категориям времени. Снимите этот флажок и убедитесь, что на рисунке останется информация, касающаяся исключительно состояний потоков.

После этого верните все в исходное положение, в котором установлены флажки **Critical Path Data** и **Concurrency**, а флажок **Behavior** неактивен. В этой конфигурации

легенды пользователь имеет возможность анализировать, насколько эффективно его приложение использует ядра процессора (уровень параллелизма).

В нашем случае можно увидеть, что основную часть критического пути занимает оранжевый цвет, что означает, что большую часть времени приложение выполнялось в последовательном режиме. Это тревожный симптом, который может означать недостаточно высокую степень распараллеливания или неравномерное распределение нагрузки между потоками.

Легенда также несет информационную функцию. Если Вы забыли, что означает тот или иной цвет, наведите курсор мыши на прямоугольник соответствующего цвета в легенде. Появится всплывающая подсказка, содержащая информацию о том, что он означает. Кроме того, если цвет считается «проблемным» (свидетельствует о неправильной организации приложения), то в подсказке будут содержаться советы для решения данной проблемы.

На легенде неизученной осталась последняя кнопка-флажок **Behavior**. Установите ее и снимите флажок **Concurrency**. В этой конфигурации пользователь имеет возможность определить поведение потока в его активном состоянии. В нашем случае критический путь окрашен в основном в красный цвет, что означает, что происходило ожидание завершения работы некоторых потоков. Это дает нам мало новой информации, поскольку мы и раньше знали, что основной поток приложения большую часть времени ожидает завершения дочерних потоков.

Установите флажки **Concurrency** и **Behavior**, чтобы получить комбинацию двух этих режимов. Тогда критический путь приобретет более разнообразную окраску. Предназначен этот режим для одновременного анализа всей совокупности характеристик поведения потоков.

### Использование функции группировки

Данная функция очень удобна при анализе критического пути, она позволяет взглянуть на оптимизируемое приложение с различных точек зрения. Так, если осуществить группировку по потокам, то можно проанализировать эффективность работы каждого из потоков. Если же включить группировку по объектам, то становится доступной информация об эффективности работы с конкретным объектом (примитивом синхронизации, например).

Рассмотрим наиболее типичные варианты использования группировки. Полезно начать с варианта, когда группировки не используются. Нажмите кнопку с изображением перечеркнутого красного круга – при этом критический путь примет вид как показано на рисунке 14.

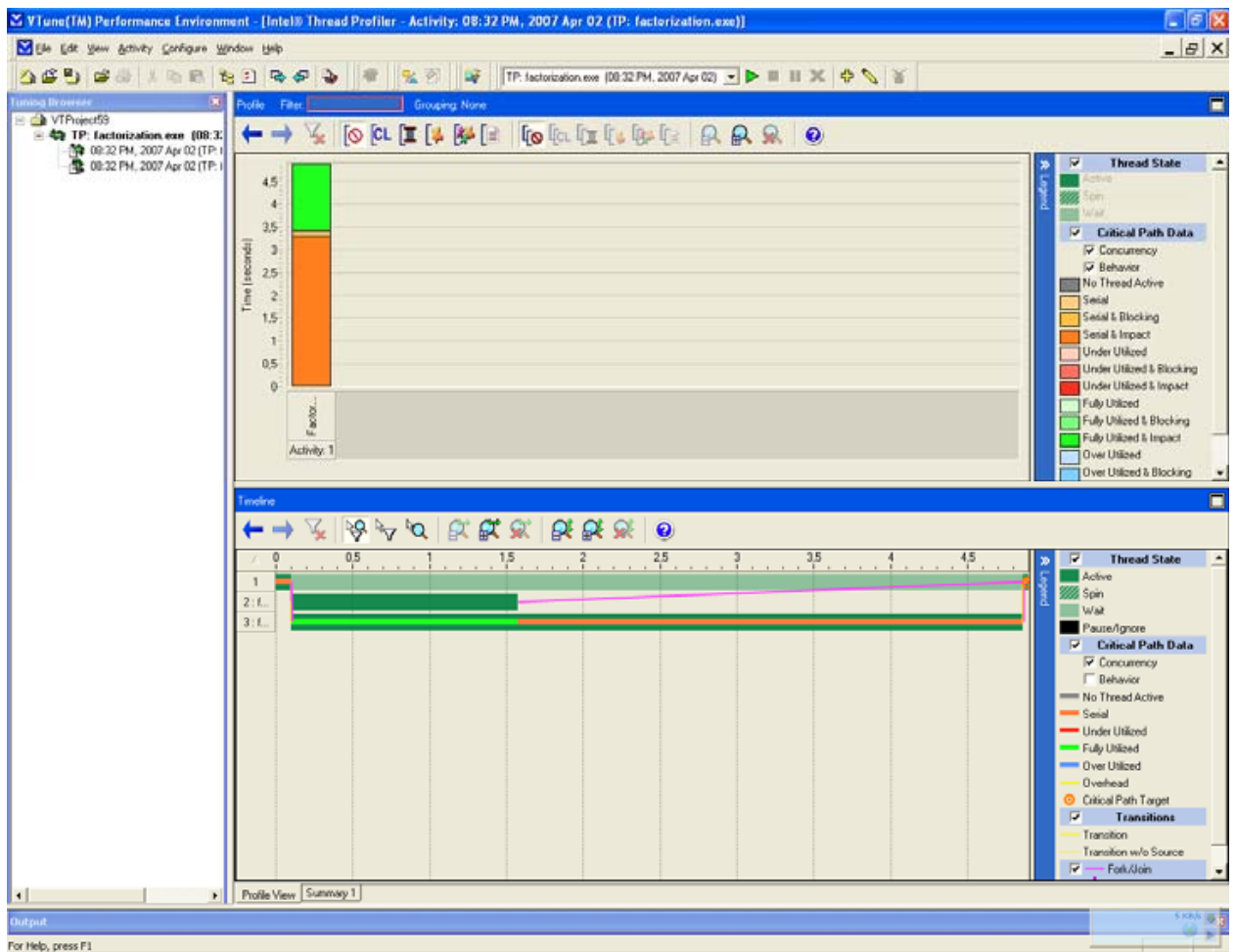


Рисунок 14 - Изображение критического пути при отключенной группировке

Этот режим наиболее полезен для получения информации о распределении времени в критическом пути. Используя его можно получить информацию о том, сколько процентов занимает последовательное выполнение (доля оранжевого цвета), сколько параллельное (зеленый, синий и красный цвета), а также какую часть занимают непроизводительные расходы (доля желтого цвета). Попробуйте также другие режимы группировки, анализируя каждый раз изменения критического пути.

Кроме того, может оказаться полезной возможность вторичной группировки.

В первом наборе кнопок группировки нажмите кнопку с изображением катушки, а во втором наборе – с буквами CL. При этом критический путь примет вид, как показано на рисунке 15.

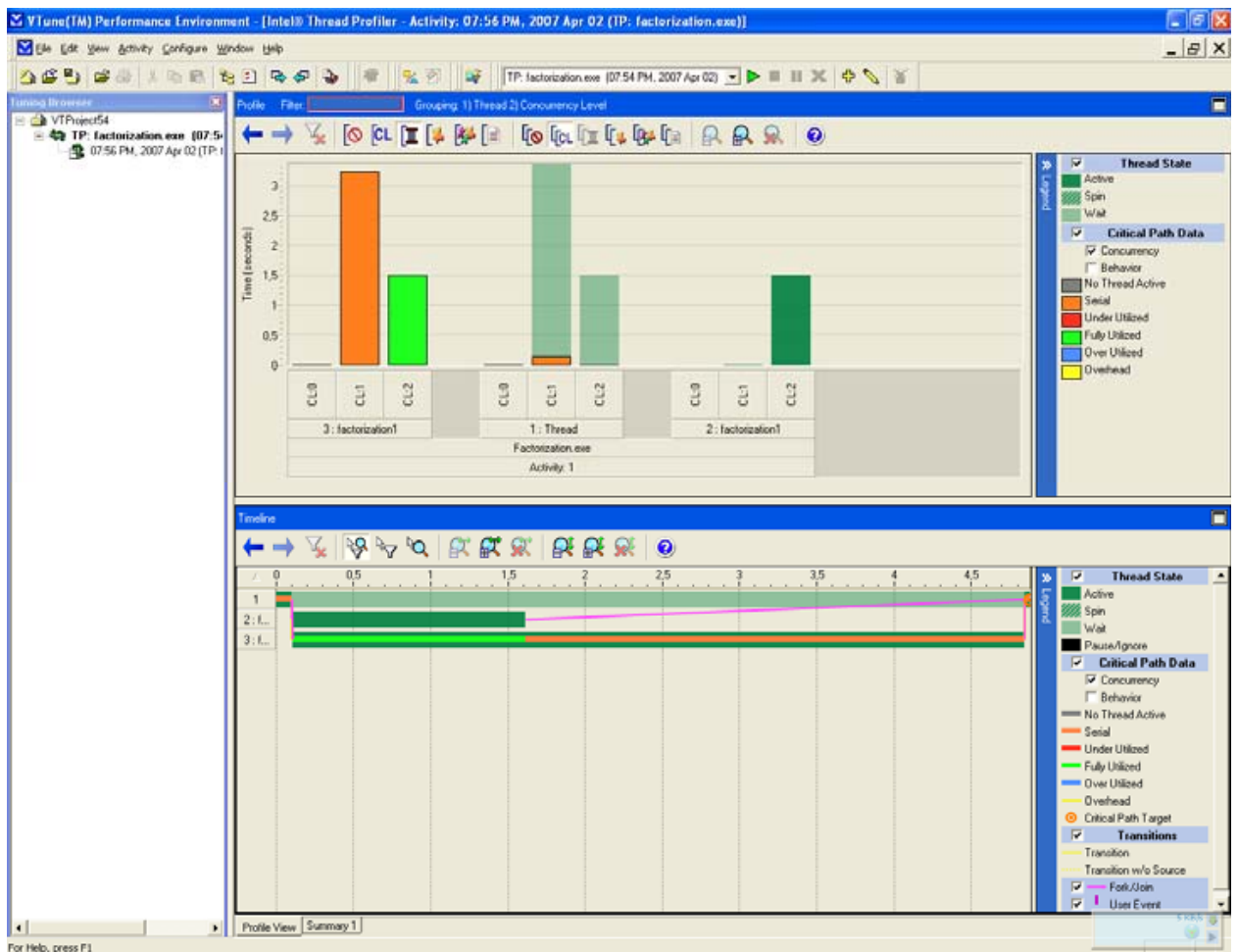


Рисунок 15 - Использование вторичной группировки

В данном случае мы имеем возможность исследовать, насколько эффективно функционировал каждый из потоков. Мы видим три группы столбиков, каждая из которых соответствует одному потоку. Столбики показывают сколько времени каждый из потоков функционировал в последовательном режиме, сколько в параллельном, а сколько в состоянии ожидания. Как можно увидеть из рисунка, третий поток (левая группа столбиков) большую часть времени работал в последовательном режиме, второй поток (правая группа столбиков) практически все время работал параллельно с третьим потоком. Основным же поток приложения (центральная группа столбиков) выполнялся лишь в последовательном режиме и большую часть жизни находился в режиме ожидания дочерних потоков.

Поэкспериментируйте, определяя различные порядки группировки, попытайтесь представить ситуации, в которых они могут быть полезны.



## Рабочая область

Мы изучили различные способы представления критического пути в рабочей области окна **Profile** и пришло время изучить способы его анализа.

Выключите группировку, чтобы распределение критического пути по категориям времени было представлено в виде одного столбца как на рисунке 16. Кроме того, установите все флажки на легенде. После этого наведите курсор на каждый из участков критического пути с различными цветами. В появляющихся подсказках содержится информация об участке критического пути, на который указывает курсор мыши.

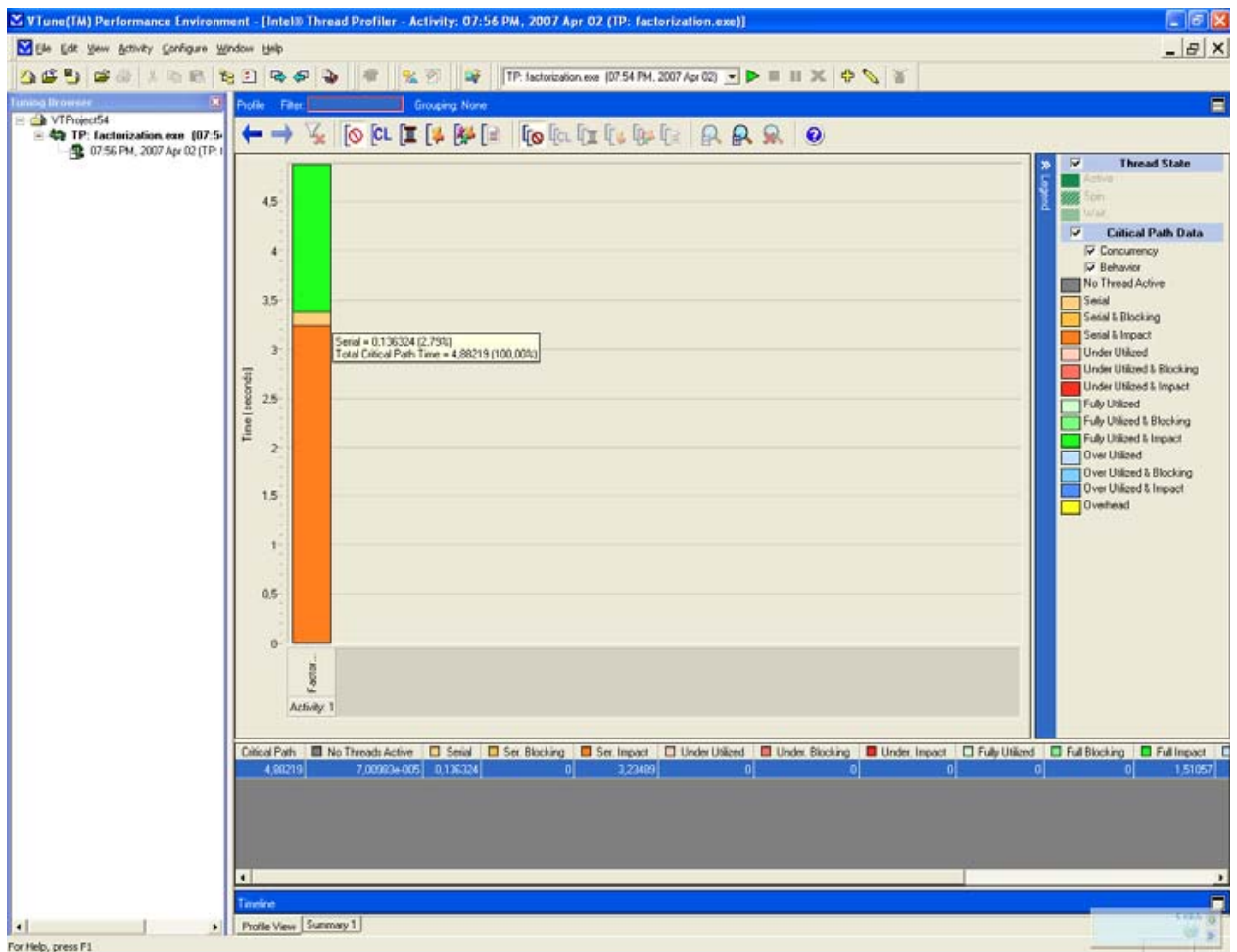


Рисунок 16 - Анализ критического пути

Наведите курсор мыши на столбец и щелкните левой кнопкой один раз. В нижней части окна **Profile** появится информация о том, сколько времени приложение функционировало в различных режимах – распределение времени по категориям.

Двойной щелчок на столбце позволяет получить более подробную информацию. Попробуем, например, получить полную информацию об участке критического пути оранжевого цвета (здесь и далее имеется в виду ярко-оранжевый цвет). Для этого наведите на него курсор и произведите двойной щелчок левой кнопкой мыши. Критический путь распадется на уровни параллелизма (concurrency), в чем можно убедиться по нажатой

кнопке CL на панели группировки и надписям под различными столбцами. Щелкните по оранжевому участку еще раз, и единый столбец снова распадется на несколько – уже по числу потоков. Еще один двойной щелчок приведет нас к уровню функций. Теперь надпись под столбцом однозначно указывает на имя функции, время работы которой вошло в критический путь с оранжевым цветом. Если произвести еще один двойной щелчок, то откроется вид исходного кода рисунку 17.

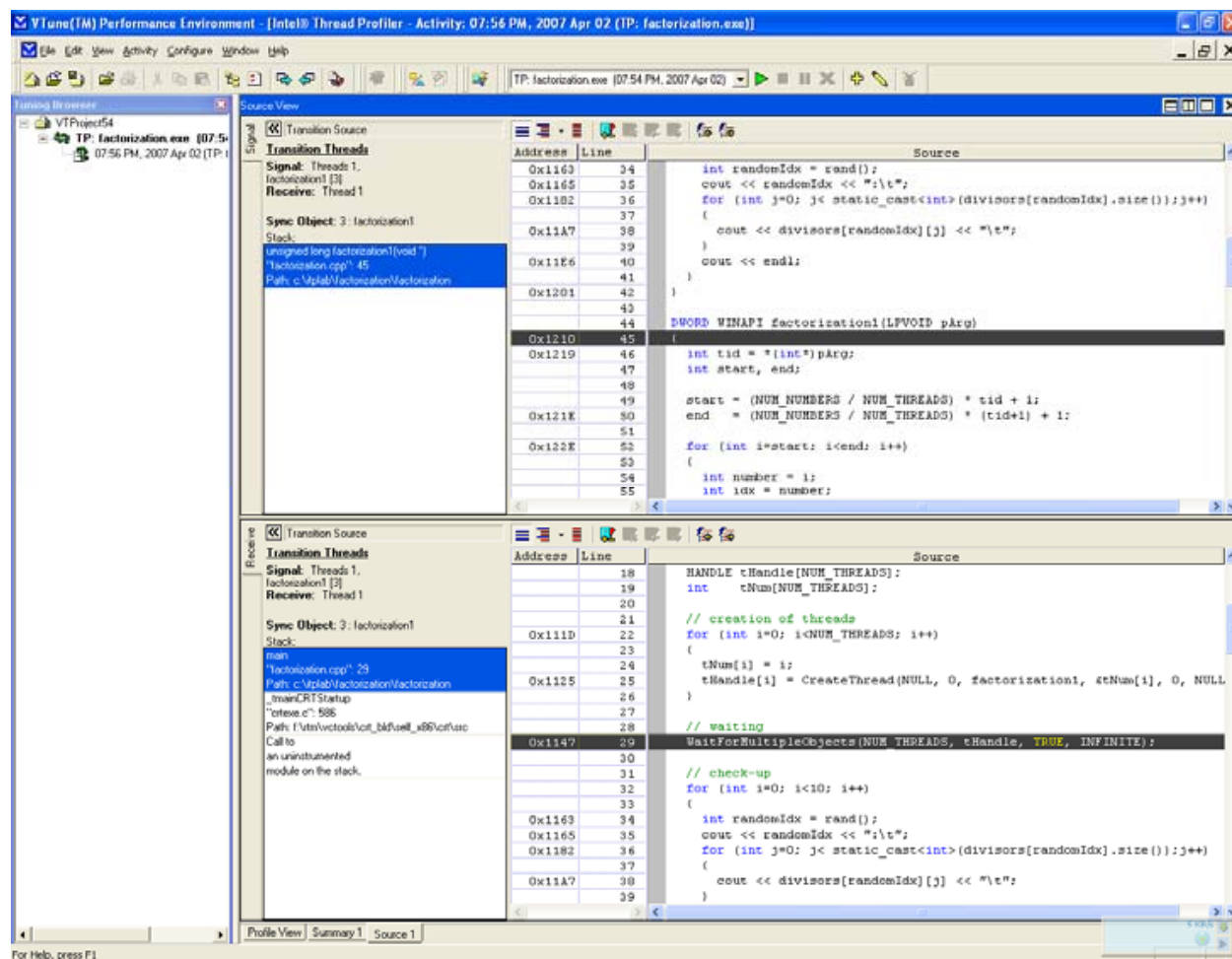


Рисунок 17 - Обращение к исходному коду приложения

Здесь мы можем убедиться, что за участок критического пути оранжевого цвета отвечает рабочая функция потоков, производящих факторизацию чисел.

Чтобы вернуться к исходному представлению критического пути, выключите все фильтры, нажав кнопку с изображением воронки, а затем отмените всякую группировку, нажав кнопку с изображением перечёркнутого круга.

Если имеется необходимость понять, какой участок кода отвечает за некоторый участок критического пути, можно поступить более простым способом, чем мы делали это раньше. А именно, выберите способ группировки по исходному коду. Для этого нажмите на кнопку с изображением файла. После этого наведите курсор мыши на интересующий вас участок критического пути и нажмите правую кнопку мыши.



Если ИТР имеет доступ к коду, который отвечает за данный участок критического пути, то в контекстном меню будет доступен пункт **Transition Source View**. Выберите его, и откроется вид исходного кода.

Однако не всегда можно спуститься до уровня исходного кода – так случается, если в приложении не содержится отладочная информация. Типичная ситуация – приложение использует вызовы библиотеки, которая была получена в скомпилированном виде.

Итак, окно **Profile** предоставляет возможности анализа критического пути. С его помощью можно узнать распределение времени работы приложения по категориям. Но при этом мы получаем очень мало информации о динамике работы приложения. Для понимания того, что происходило с потоками во время исполнения, как они взаимодействовали между собой, обратимся к следующему окну ИТР – окну **Timeline**.

### Окно Timeline

Окно Timeline содержит три основных элемента, также как и окно **Profile**: рабочая область, панель инструментов и легенда. Панель инструментов в данном случае предназначена лишь для изменения масштаба временной оси.

#### Рабочая область

В рабочей области окна **Timeline** содержится информация о поведении потоков. В верхней части рабочей области имеется временная шкала. Ниже располагаются полосы, каждая из которых соответствует одному потоку приложения.

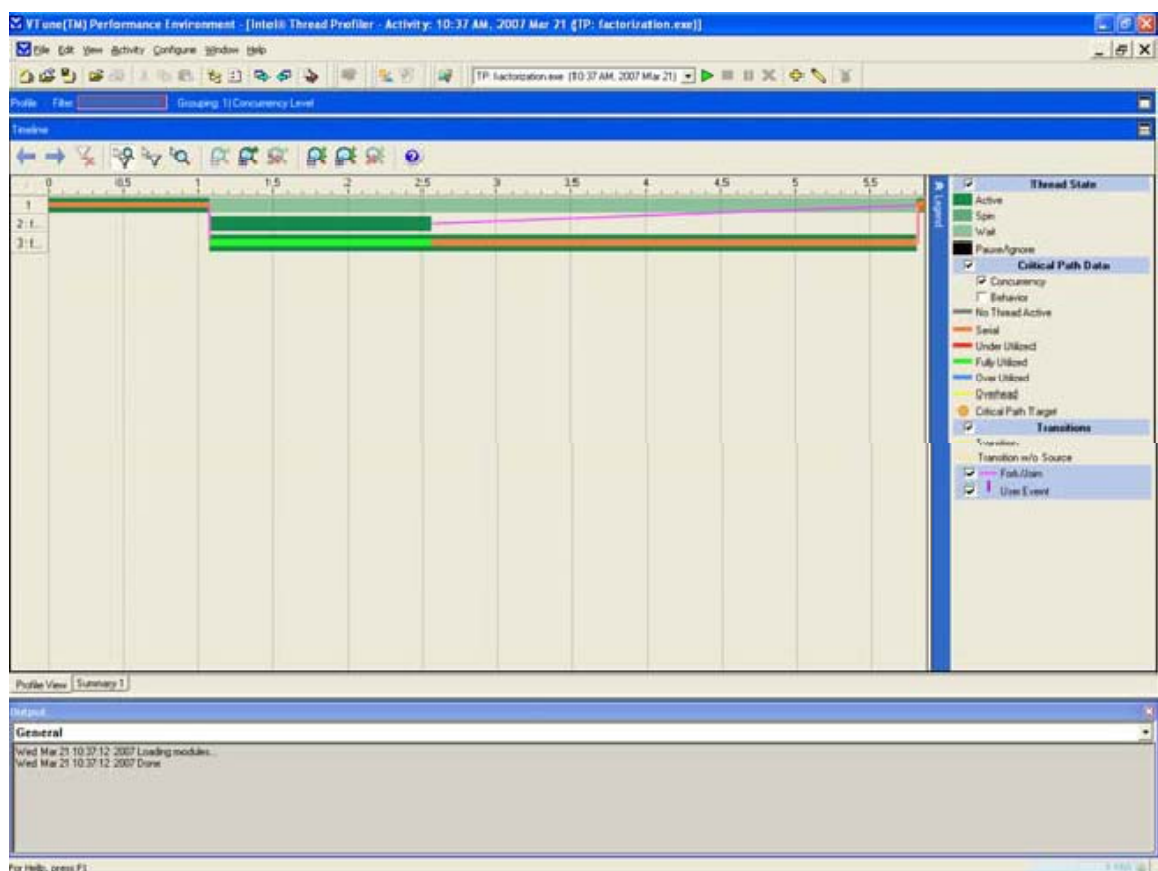


Рисунок 18 - Окно Timeline

На рисунке 18 показано, что в нашем приложении было запущено три потока. Также можно увидеть, что один из них существовал с начала времени выполнения приложения, а два других были созданы позже.

Время жизни потока равно длине полосы. Раскраска полосы не всегда одинакова – она указывает на состояния потока в различные моменты времени (см. легенду). Наведите курсор мыши на бледно-зеленую часть самой верхней полосы, соответствующей основному потоку. В появившейся подсказке будет сказано, что на этом промежутке времени поток находился в режиме ожидания.

Рассмотрим подробнее момент возникновения потоков. Для этого в рабочей области необходимо выделить область, охватывающую розовую стрелку. Наведите курсор мыши немного левее стрелки, нажмите левую клавишу и, передвиньте курсор правее стрелки, после чего отпустите кнопку мыши. Повторите увеличение еще несколько раз, пока вид в рабочей области не станет таким, как показано на рисунке 19.

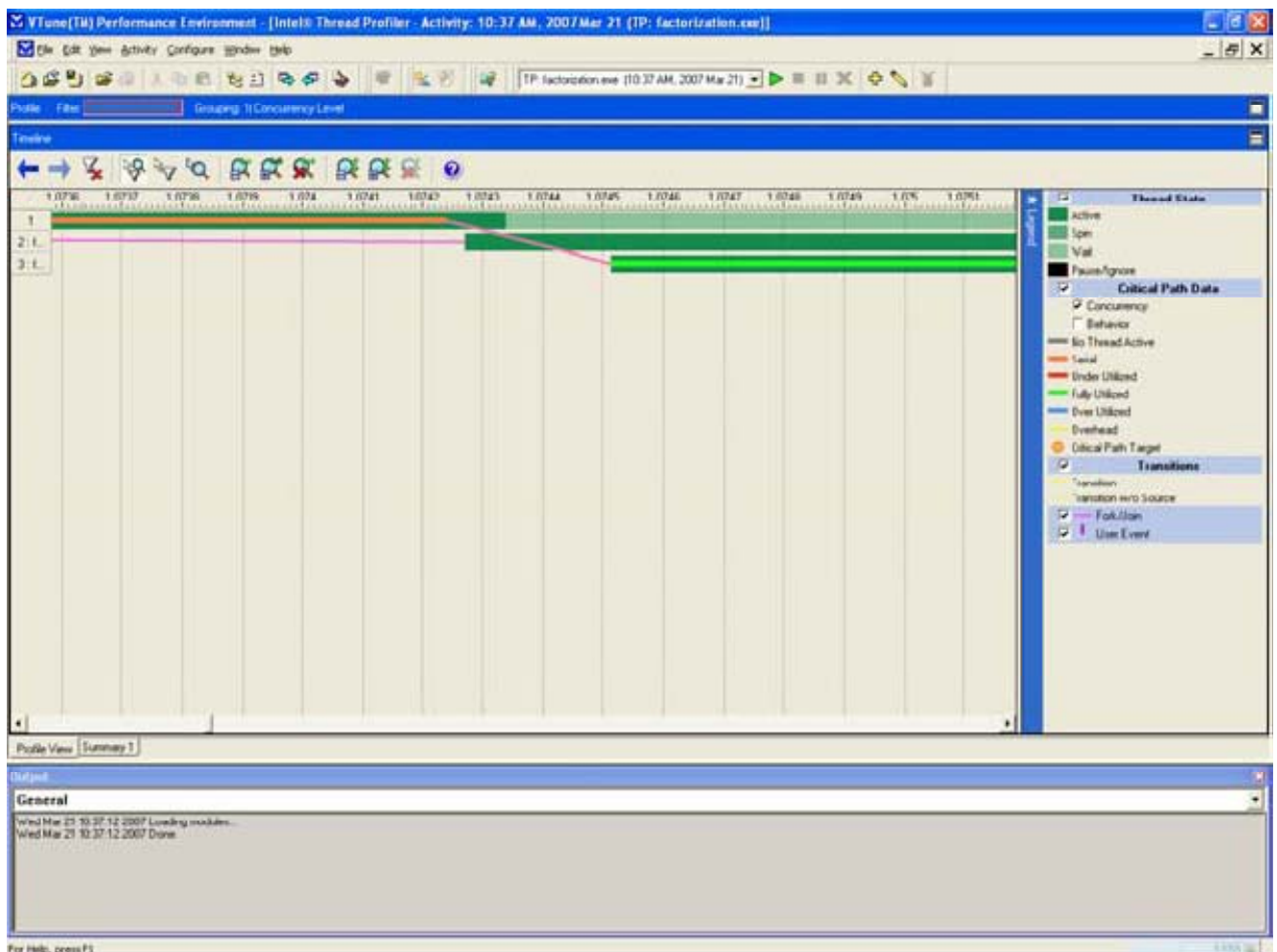


Рисунок 19 - Момент создания потоков

Здесь уже можно видеть, что второй поток был создан несколько позже первого. Наведите курсор мыши на одну из розовых стрелок и убедитесь, что она соответствует вызову функции создания потока.

Длина этой стрелки, спроектированная на ось времени, показывает время задержки между вызовом функции создания потока и фактическим его запуском.

Кроме того, поверх зеленых полос, указывающих состояние потоков, яркими цветами накладывается информация о критическом пути. На изучаемом нами промежутке времени имеются участки последовательного и параллельного исполнения потоков. Нетрудно понять, что оранжевый участок соответствует интервалу времени, когда в приложении существовал только один поток, а участок зеленого цвета соответствует одновременной работе двух потоков.

Вернемся к изучению трассы приложения в целом. Нажмите на панели инструментов окна **Timeline** кнопку с изображением воронки и красного креста. Рабочая область снова должна принять вид как на рисунке 18.

Глядя на этот рисунок, можно сразу понять, в чем причины того, что основную часть времени наше приложение работает в последовательном режиме. Мы неравномерно распределили вычислительную нагрузку между потоками. Второй из дочерних потоков работает гораздо дольше первого, в результате чего увеличивается суммарное время работы приложения. Таким образом, если мы хотим повысить производительность нашего приложения, то первое, что мы должны сделать, это распределить нагрузку между потоками равномерно. Тогда время работы приложения сократится за счет того, что часть нагрузки второго дочернего потока будет отдана первому.

Последнее, что осталось изучить в рабочей области окна **Timeline** – это как из него получить доступ к исходному коду. Наведите курсор мыши на любую из полос, соответствующих дочерним потокам, и нажмите правую кнопку мыши. В появившемся контекстном меню выберите пункт **Thread Creation/Entry Source View**. Откроется окно с исходным кодом, отвечающим за создание потока.

Вернитесь к окну **Timeline** и наведите курсор на бледно-зеленую часть полосы, соответствующей основному потоку приложения, нажмите правую кнопку мыши и выберите пункт **Transition Source View**.

Откроется окно с исходным кодом, в котором указано место ожидания основного потока (рисунок 20).

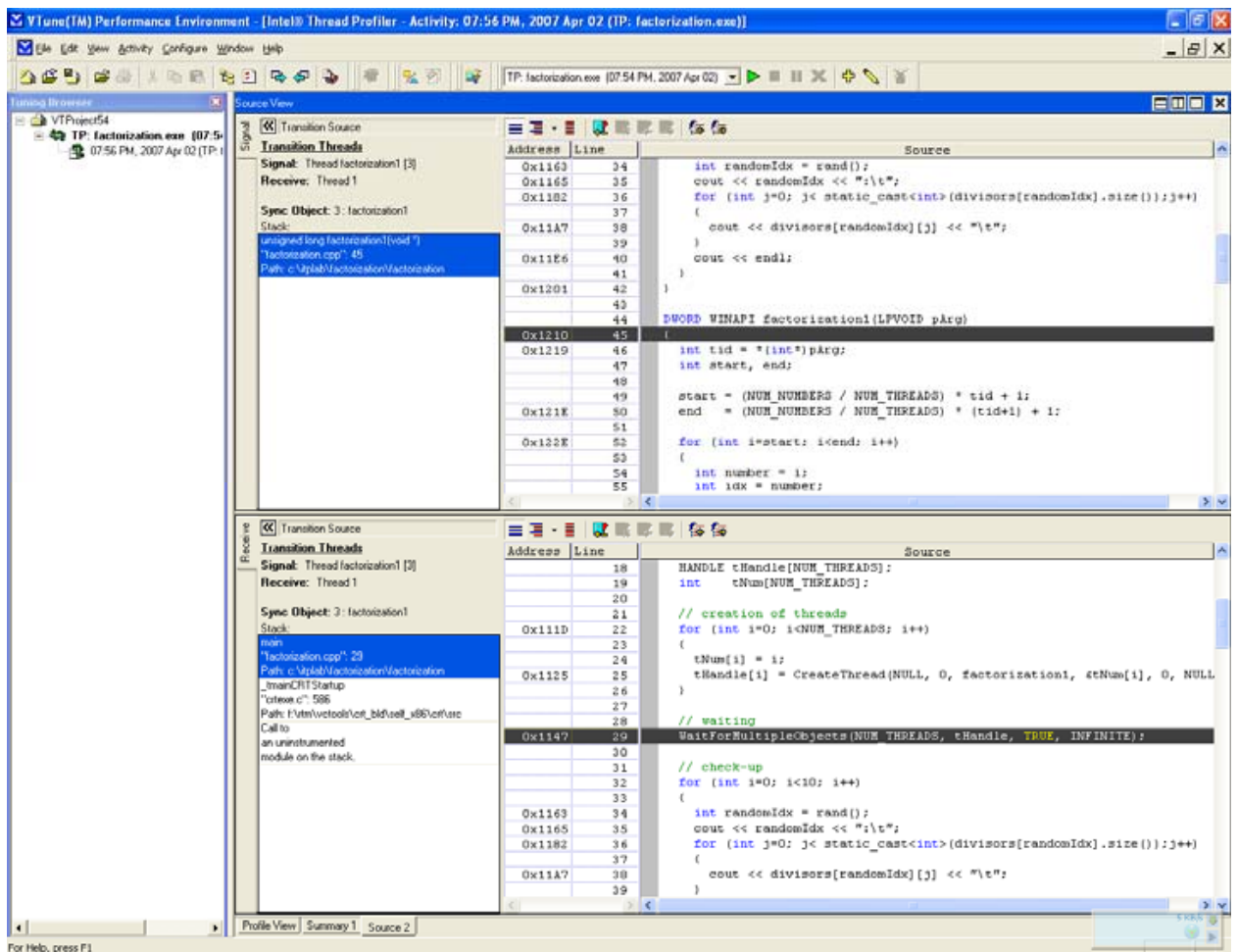


Рисунок 20 - Место ожидания основного потока

В нашем случае это вызов функции **WaitForMultipleObjects**, которая необходима здесь для ожидания завершения дочерних потоков.

Вернемся к окну **Timeline** и рассмотрим легенду.

### Выбор показателей поведения потоков для визуализации

Общая структура и функции легенды такие же, как и в окне **Profile**, поэтому мы не будем останавливаться на кнопках-флажках **Thread State** и **Critical Path Data**. Вы можете поэкспериментировать с ними и проследить, как меняется вид в рабочей области окна **Timeline**.

Рассмотрим назначение новых кнопок-флажков. Первый из них имеет название **Transitions** и отвечает за отображение в рабочей области стрелок, соответствующих посылке сигналов между потоками. Снимите флажок **Fork/Join** – останутся три стрелки желтого цвета, показывающие посылаемые в приложении сигналы. В нашем случае это сигналы, связанные с созданием и завершением потоков.

Теперь снимите флажок **Transitions** и установите флажок **Fork/Join**. Должны появиться стрелки, указывающие на вызовы функций класса **Fork/Join** (создание и ожидание завершения потоков). Можно заметить, что они совпадают с желтыми

стрелками, которые мы видели до этого. Единственное отличие – появление розовой стрелки, соединяющей конец полосы первого дочернего потока с полосой основного потока.

Последний флажок называется **User Event**. Мы не станем рассматривать его, подробная информация о нем может быть найдена в [7].

Таким образом, мы выяснили, что наше приложение содержит поток, большую часть времени работающий в последовательном режиме. Причины такой ситуации и способы увеличения производительности нашего примера мы рассмотрим в соответствующей лабораторной работе.

### **Контрольные вопросы**

1. Наличие каких цветов в критическом пути свидетельствует о проблемах с производительностью приложения?
2. На какие цвета, по вашему мнению, нужно обратить внимание прежде всего?
3. Если вам известно несколько причин низкой производительности вашего приложения, в какой последовательности лучше их устранять?
4. Зачем используется группировка?
5. В чем разница между группировкой по объектам и по типам объектов?
6. В окне Profile установите первичную группировку по потокам, а вторичную по уровню параллелизма. Установите соответствие между столбцами в окне Profile и полосами в окне Timeline.
7. Мы установили, что основная причина медленной работы нашего приложения – неравномерное распределение нагрузки между потоками. Предложите ваш способ решения данной проблемы.

### **Литература**

1. «Developing Multithreaded Applications: A Platform Consistent Approach», Intel Corporation, March 2003.
2. «Threading Methodology: Principles and Practices», Intel Corporation, March 2003.
3. «Multi-Core Programming for Academia», Student Workbook, by Intel.
4. «Multi-Core Programming», book by Sh. Akhter and J. Roberts, Intel Press 2006.
5. «Intel® Thread Profiler. Getting Started Guide».
6. «Intel® Thread Profiler. Guide to Sample Code».
7. «Intel® Thread Profiler Help».