Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию для многоядерных компьютерных систем

Учебный курс «Технологии разработки параллельных программ»

Раздел «Оптимизация параллельной программы»

Intel Thread Profiler – краткое описание

Разработчики: К.В. Корняков, А.В. Шишков

Нижний Новгород
2007

Содержание

1. Введение

Многопоточный подход к разработке приложений появился в программировании уже достаточно давно и широко используется для повышения производительности. С появлением многоядерных процессоров интерес к многопоточному программированию значительно усилился. Многоядерные системы прошли путь от лабораторных образцов до настольных компьютеров, доступных рядовым пользователям. В 2005 году началась массовая продажа процессоров с двумя ядрами, в 2007 – четырехъядерных процессоров. В будущем ожидается сохранение тенденции увеличения числа ядер, и можно с уверенностью сказать, что с течением времени число систем, построенных на базе многоядерных процессоров, будет постоянно увеличиваться.

В подобной ситуации остро встает вопрос об эффективном использовании многоядерных архитектур. Многопоточное программирование представляет собой очень удобный подход, позволяющий создавать приложения, способные рационально использовать процессоры с несколькими ядрами, обеспечивая их оптимальную загрузку. Уже сегодня подавляющее большинство существующих приложений использует несколько потоков: браузеры, файловые менеджеры, словари, антивирусы и многие другие. Можно ожидать, что в дальнейшем все больше приложений будет изначально разрабатываться как многопоточные.

Необходимо, тем не менее, отметить, что разработка многопоточных приложений зачастую на порядок сложнее, чем последовательных. Новая парадигма приносит не только новые возможности, но и новые сложности. Создание эффективных многопоточных приложений – весьма трудоемкий процесс, а значит, имеется насущная необходимость в инструментальной поддержке всех стадий разработки подобных приложений, начиная с построения архитектуры и заканчивая настройкой параметров и финальной оптимизацией приложения.

В настоящее время существует достаточно много инструментов, ориентированных на поддержку многопоточного программирования. Компания Intel выделяется на общем фоне тем, что предлагает целое семейство подобных инструментов, представителем которого является Intel® Thread Profiler (ITP).

1.1. Семейство инструментов Intel для поддержки разработки многопоточных приложений

Цикл разработки программных приложений включает в себя следующие основные стадии: анализ требований, разработка архитектуры, реализация, тестирование, отладка и оптимизация. Зачастую при этом приходится проходить несколько раз по каждой из стадий. Эта деятельность разработчиков требует наличия специализированных инструментов, и в настоящее время компания Intel® предлагает целый ряд программных продуктов, предназначенных для поддержки разработки многопоточных приложений:

- Intel® C/C++ Compiler и Intel® Fortran Compiler – компиляторы;
- Intel® Performance Libraries – библиотеки математических функций (численные методы, обработка сигналов, генераторы случайных чисел);
- Intel® VTune Performance Analyzer – профилировщик;
- Intel® Thread Checker – отладчик многопоточных приложений;
- Intel® Thread Profiler – профилировщик многопоточных приложений;
- Intel® Threading Building Blocks – C++ библиотека времени выполнения, представляющая набор примитивов для разработки многопоточных приложений (параллельные алгоритмы, потокобезопасные структуры данных, примитивы синхронизации).

Данное семейство инструментов предназначено для поддержки всего цикла разработки многопоточных приложений. Их конечная цель – значительно ускорить процесс разработки, обеспечив разработчиков удобными и эффективными инструментами. Инструменты поддерживают различные методы создания многопоточных приложений: OpenMP, Pthreads, Win32 Threading API, автоматическое распараллеливание. Более подробную информацию можно найти на сайте http://www.intel.com/software/products.

На рис. 1 схематически представлен цикл разработки многопоточного приложения с указанием программных продуктов компании Intel, предназначенных для помощи на соответствующих стадиях.

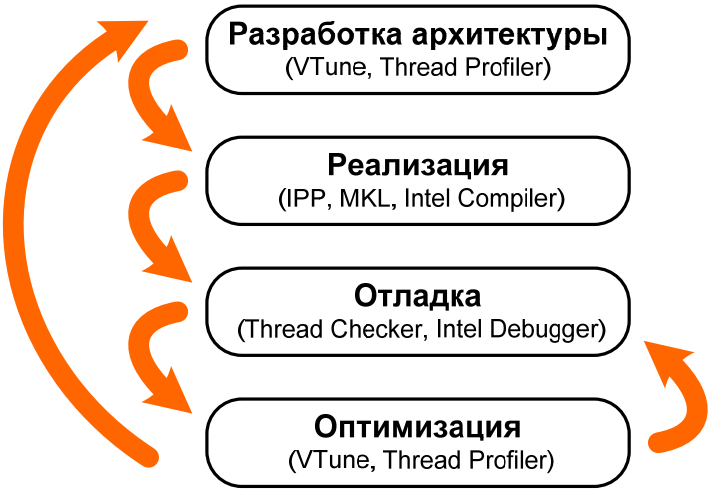

Рис. 1. Цикл разработки приложения

Как видно из диаграммы, ITP в основном призван помочь разработчикам на финальных стадиях создания многопоточного приложения. Он предназначен для профилирования многопоточных приложений с целью их последующей оптимизации и финальной настройки. Однако следует заметить, что ITP может оказаться полезным еще на стадии разработки архитектуры. Разработчик имеет возможность сравнивать между собой различные архитектурные решения, создавая прототипы и сравнивая их производительность при помощи ITP и VTune.

1.2. Определение и цели профилирования

Центральная идея ускорения последовательных приложений достаточно проста. Необходимо выявить участки кода, в которых приложение проводит основную часть времени и оптимизировать их. Известен эвристический закон «20/80», утверждающий, что приложение проводит 80% времени работы в 20% кода. Таким образом, за счет оптимизации именно этих 20% кода можно добиться значительного повышения производительности. Как раз для поиска таких узких мест используются специальные средства, называемые профилировщиками.

Под понятием *профилирование* обычно понимается процесс анализа производительности приложения и учета потребляемых им ресурсов. Тем самым, *профилировщик* – это инструмент, при помощи которого осуществляется профилирование.

Профилировщики исследуют поведение программы в процессе ее выполнения. В частности, собирается такая информация, как частота вызовов функций, продолжительность их работы, число потоков, время ожидания – любая информация, способная помочь при оптимизации производительности. В числе приемов, используемых профилировщиками для сбора информации, инструментация кода, аппаратные прерывания, внедрение счетчиков производительности и многие другие. Выходной информацией профилировщика является *трасса* – список событий, произошедших в приложении (исполненные инструкции, вызовы функций, вызовы операционной системы). После получения трассы профилировщик обрабатывает собранную информацию и строит *профиль* приложения – статистическую сводку о работе приложения. Затем профиль представляется в графическом виде, удобном для анализа. Разработчик изучает его, определяет проблемные места и ищет способы оптимизации своего приложения.

Ситуация с многопоточными приложениями существенно другая, поскольку в силу вступают новые аспекты производительности, связанные с взаимодействием потоков, затратами на их создание и синхронизацию. Ускорение отдельных участков кода может не дать никакого прироста производительности, если, например, разработчик построил свое приложение таким образом, что основную часть времени потоки ожидают освобождения разделяемого ресурса. Также можно упомянуть такие распространенные в многопоточных приложениях проблемы как неравномерное распределение нагрузки и неэффективное использование примитивов синхронизации. Эти факторы могут привести к катастрофически низкой производительности приложения, вплоть до того, что многопоточная версия будет медленнее последовательной.

Таким образом, для анализа новых аспектов программной оптимизации, присущих именно многопоточным приложениям, необходимы специальные инструменты, ориентированные на обнаружение проблемных ситуаций и последующей помощи программистам в их разрешении. Одним из самых эффективных и популярных представителей инструментов такого класса является ITP.

1.3. Назначение Intel Thread Profiler

ITP предназначен для профилирования многопоточных приложений и помощи разработчику в процессе оптимизации программного кода. Следует напомнить, что ITP также может быть использован на стадии разработки архитектуры приложения. Разработчики могут создать несколько прототипов, соответствующих различным архитектурным решениям, и сравнить их производительность при помощи ITP. Также ITP может быть использован для анализа масштабируемости приложения. Например, если заранее известны характеристики компьютерной системы (число процессоров или ядер), на которой будет функционировать приложение, то ITP можно использовать для подбора оптимального числа потоков.

Итак, ITP может оказать разработчикам существенную помощь при решении проблем, связанных с производительностью многопоточного приложения. Перечислим основные варианты использования Intel® Thread Profiler:

- анализ эффективности распределения вычислительной нагрузки между потоками (подраздел 4.1, лабораторная работа 1);
- анализ эффективности обращения к разделяемым ресурсам (подраздел 4.2, лабораторная работа 2);
- определение наиболее медленных потоков и, как результат, нуждающихся в оптимизации (лабораторная работа 1);
- выбор оптимальной архитектуры многопоточного приложения (число потоков, алгоритм, примитивы синхронизации) (лабораторная работа 2);
- анализ эффективности работы с потоками и примитивами синхронизации (лабораторная работа 2);
- анализ масштабируемости приложения на вычислительных узлах с различным числом процессоров.

Таким образом, ITP является весьма удобным и эффективным инструментом для поиска причин низкой производительности многопоточного приложения и последующей помощи в их устранении.

2. *Технические характеристики Intel Thread Profiler*

2.1. Поддерживаемые методы создания многопоточных приложений

ITP позволяет профилировать многопоточные приложения, созданные на основе технологий OpenMP, Windows threads, POSIX threads.

Инструмент имеет два режима: «OpenMP» и «Threaded». Первый из них используется для анализа приложений, созданных с использованием OpenMP и скомпилированных компиляторами компании Intel. Этот режим предлагает возможности сбора и отображения следующей информации:

- затраты на синхронизацию, накладные расходы на поддержку работы с потоками;
- дисбаланс вычислительной нагрузки;
- сравнение результатов разных запусков.

Главной особенностью этого режима является возможность оценивания потенциальной масштабируемости, что очень удобно при создании параллельного прототипа приложения.

Второй режим является более общим – он обеспечивает анализ OpenMP приложений и многопоточных приложений, основанных на использовании Windows threads или POSIX threads. В этом режиме отображается следующая информация:

- аналогичные «OpenMP» режиму данные по потокам программы и уровню паралеллизма;
- критический путь программы;
- распределение временных затрат по критическому пути на:
  - исполнение;
  - синхронизацию;
  - ожидание;
  - блокировку.
- связь потоковых событий с исходным кодом.

Режим «Threaded» предоставляет более богатые возможности, поэтому мы сосредоточимся на рассмотрении именно этого режима.

2.2. Системные требования

2.2.1. Аппаратное обеспечение

Минимальные требования:

- Процессор Intel® Pentium 4;
- ОЗУ 512 МБ;
- Свободное дисковое пространство 300 МБ.

При практическом использовании ITP для достижения приемлемых показателей оперативности работы могут быть рекомендованы повышенные требования к минимально-необходимым аппаратным ресурсам:

- Процессор Intel® Pentium 4, поддерживающий технологию Hyper-Threading, или процессор Intel® Xeon;
- ОЗУ 2 ГБ.

Для изучения всех аспектов "реальных" параллельных вычислений желательно использование многоядерных процессоров компании Intel.

2.2.2. Программное обеспечение

Минимальные требования:

- Microsoft® Windows® XP Professional, или Microsoft® Windows® Server 2003.
- Microsoft® Visual C++® 6.0 или выше.
- Microsoft® Internet Explorer® 6.0 или выше.
- Adobe® Acrobat® Reader.

Для анализа OpenMP-приложений и инструментации на уровне исходных кодов требуется следующее программное обеспечение:

- Intel® C++ Compiler 8.1 или выше для Windows® для архитектуры IA-32;

- Intel® C++ Compiler 9.1 или выше для Windows® для архитектуры Intel® EM64T;
- Intel® Fortran Compiler для Windows® 8.1, Package ID: w_fc_pc_8.1.023 или выше.

3. Основные концепции и понятия профилирования

3.1. Понятие критического пути

Первое понятие, которое нам понадобится – *критический путь (critical path)*. Это понятие является ключом к пониманию всего процесса профилирования, поэтому очень важно хорошо усвоить его. Прежде чем дать формальное определение этого понятия, попытаемся понять его суть на примере.

Представим себе следующее многопоточное приложение: имеется основной поток, который подготавливает данные и производит их первичную обработку, а также один дочерний поток, который завершает обработку и выводит результаты. Простейшая иллюстрация этого примера – физическое моделирование, в котором основной поток занят вычислениями (например, решением разностной схемы), а дочерний – подготовкой к визуализации и непосредственно визуализацией результатов вычислений. Часто подобные взаимодействия реализуются в виде конвейера, поскольку в то время как один из потоков работает с устройствами ввода-вывода, второй поток может нагружать процессор вычислениями.

В нашем случае, как только первый поток вычислил первую порцию данных для отображения, он может начинать вычисление следующей порции, а второй поток параллельно с этим начнет визуализацию. Важно при этом обеспечить синхронизацию между потоками, чтобы не допустить уничтожения данных до того как они были визуализированы.

Ниже графически представлено взаимодействие потоков в случае, если бы требовалось выполнить только две итерации. Потокам в нашем приложении соответствуют горизонтальные линии, сплошные участки соответствуют активному состоянию потоков, а пунктирные – состоянию ожидания. Диагональные линии соответствуют сигналам, передаваемым между потоками, а проекции этих линий на ось времени соответствует временам прохождения сигналов. Кружками обозначены все события, в результате которых изменяются состояния потоков (создание/завершение, блокировка/активизация).

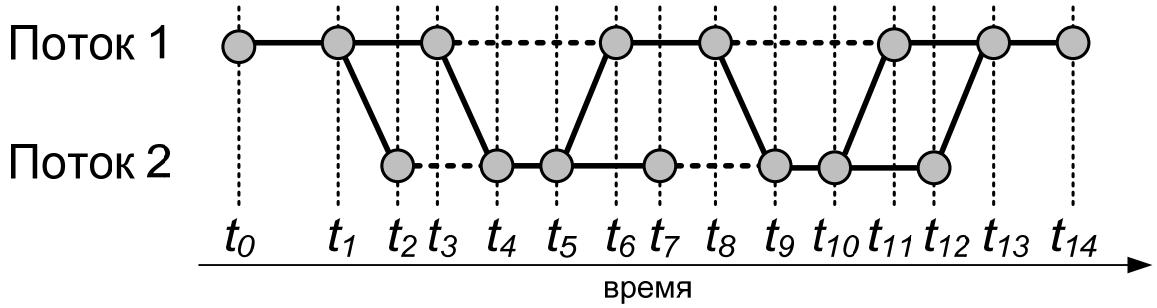

Рис. 2. Пример критического пути многопоточного приложения

Рассмотрим диаграмму подробнее. В начальный момент времени существовал только один поток, выполняющийся последовательно. Затем в момент времени t_1 происходит создание дочернего потока, который начинает функционировать в момент времени t_2, но сразу попадает в состояние ожидания, поскольку данные для визуализации еще не подготовлены. Затем в точке t_3 завершаются вычисления, и дочерний поток начинает визуализацию подготовленных данных, которая продолжается с момента t_4 до момента t_7. Заметим, что второй поток может освободить массивы с результатами вычислений сразу после того, как переведет их в формат, необходимый для графического представления (момент времени t_5). За счет этого и достигается распараллеливание в рассматриваемом приложении. Можно видеть, что потоки работают параллельно на промежутке времени между моментами t_6 и t_7. Далее осуществляется еще одна итерация, после чего в момент времени t_{12} второй поток завершает свое исполнение, а в момент времени t_{14} завершается весь процесс.

Рассмотренная диаграмма представляет собой граф. Вершинам, как уже говорилось, соответствуют события, изменяющие состояние потоков, а ребрам – действия, выполняемые потоками (мы не учитываем пунктирные линии). При этом ребрам можно сопоставить вес, равный времени, затрачиваемому на выполнение действия. *Путем приложения* называется любой путь в указанном графе, соединяющий вершины, соответствующие началу и завершению приложения. *Критическим путем приложения* называется путь, имеющий максимальную сумму весов входящих в него ребер. Приложения со многими потоками могут иметь большое число путей, однако в нашем случае приложение имеет всего два пути, каждый из которых можно назвать критическим, так как суммы их весов совпадают.

Важность понятия критического пути определяется тем, что *сумма весов ребер в критическом пути равна времени выполнения приложения*. Из этого факта следует следующий принцип: необходимо оптимизировать прежде всего те участки приложений, которые вошли в критический путь, поскольку именно они определяют производительность.

Таким образом, профилирование и оптимизация приложения неразрывно связаны с процессом анализа критического пути. Критический путь – это первое, на что нужно обращать внимание, поскольку не входящие в него участки приложения вносят меньший вклад в суммарное время работы приложения. Далее мы увидим, что ITP позволяет строить критический путь приложения и предоставляет различные средства для его анализа.

3.2. Состояния потоков

При анализе критического пути пользуются таким понятием как *состояние потока (thread state)*. А именно, поток может находиться в трех состояниях:

- *Активное состояние (active)* – поток исполняется в настоящее время;
- *Состояние активного ожидания (spin)* – постоянно повторяемая проверка состояния блокировки (например, при использовании `pthread_spin_lock()`);
- *Состояние ожидания (wait)* –ожидание завершения какой-либо блокирующей операции (например, операции ввода-вывода).

3.3. Понятие категорий времени

Рассмотрим теперь характеристики эффективности использования аппаратных ресурсов активными потоками приложения. Для этого в ITP используются два параметра: *уровень параллелизма (concurrency)* и *поведение (behavior)*.

Уровни параллелизма используются для обозначения того, насколько полно потоки нагружают процессоры вычислительного узла. Так, одновременная работа двух потоков на двухъядерном процессоре означает эффективное использование его ресурсов (*full utilization*), на одноядерном – сверхиспользование (*over utilization*), а на четырехъядерном – недостаточно эффективное использование (*under utilization*).

Существует несколько типов поведения потока в активном состоянии:

- *Сдерживание (impact)* – ситуация, когда поток сдерживает выполнение другого потока (например, поток занял мьютекс, освобождения которого ожидают другие потоки);
- *Блокировка (blocking)* – приостановка потока в результате вызова блокирующих функций (например, операций ввода/вывода);
- *Критический путь (critical path)* – ситуация, когда поток выполняет участок кода, входящий в критический путь, при отсутствии других ожидающих потоков.

Уровень параллелизма и поведение независимы друг от друга, и в совокупности они называются *категориями времени (time categories)* и характеризуют эффективность работы активных потоков. Собственно весь анализ производительности состоит в том, чтобы оценить каждый участок критического пути по этим двум параметрам и определить наиболее проблемные участки приложения.

В ITP используется несколько видов представления критического пути, каждое из которых мы впоследствии подробно изучим. Для удобства анализа каждый участок критического пути окрашивается в определенный цвет в зависимости от уровня параллелизма и поведения потока на этом участке. На рис. 3 представлена сводная таблица используемых цветов. *n* обозначает число активных потоков приложения, а *p* – число процессоров (ядер).

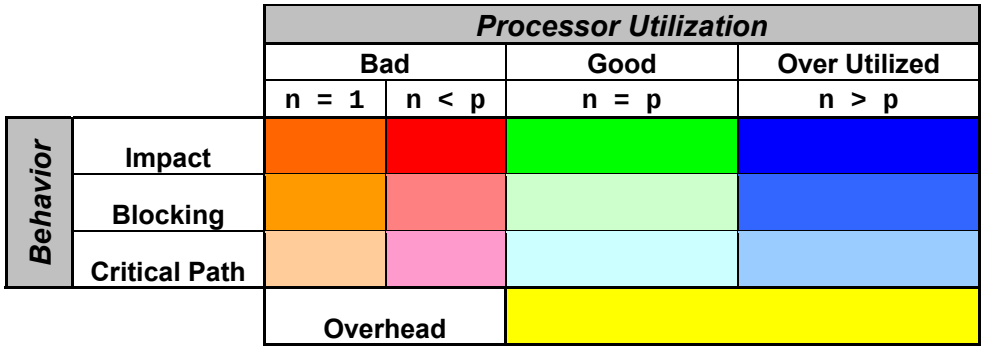

Рис. 3. Цвета, используемые для обозначения категорий времени

Рассмотрим приведенную таблицу подробнее, чтобы знать впоследствии, на какие цвета нужно обращать внимание прежде всего. Сначала исследуем связь числа потоков и числа ядер. Как видно из таблицы, для обозначения этого соотношения используется четыре цвета: оранжевый, красный, зеленый и синий. Зеленый цвет – признак оптимального соотношения, когда число ядер и потоков совпадает. Синий цвет используется для обозначения ситуаций, когда одновременно работает больше потоков, чем доступно

ядер. Как результат, потоки могут мешать работе друг друга. Присутствие красного и оранжевого цветов в окраске критического пути чаще всего свидетельствует о необходимости перестройки приложения. Эти цвета означают, что используются не все ядра процессора и можно получить дополнительное ускорение за счет более эффективного распараллеливания.

Рассмотрим теперь второй параметр – тип поведения потоков. Для обозначения типа поведения потока цветом используется изменение яркости. При этом используется простое правило: чем ярче цвет участка, тем большего внимания от разработчика он требует. Самый яркий цвет используется для указания *impact*-состояния потока, поскольку прежде всего необходимо сокращать время ожидания потоков друг другом. Далее по важности следует *blocking*-состояние, которое тоже требует пристального контроля, так как нужно минимизировать время ожидания потоков. Последним идет состояние *critical path*.

Имеется особая категория времени, обозначаемая желтым цветом и указывающая на *непроизводительные издержки* (*overhead*) в многопоточном приложении. Под непроизводительными издержками понимаются накладные расходы на создание потоков, управление и синхронизацию между потоками. Другими словами, эта величина показывает количество процессорного времени, израсходованного на поддержку работы потоков. Чаще всего большая доля желтого цвета в критическом пути свидетельствует о чрезмерно частом вызове функций синхронизации и необходимости перепроектирования приложения.

Теперь, зная о категориях времени, мы можем сказать, какие цвета будут содержаться на критическом пути нашего примера из подраздела 3.1.

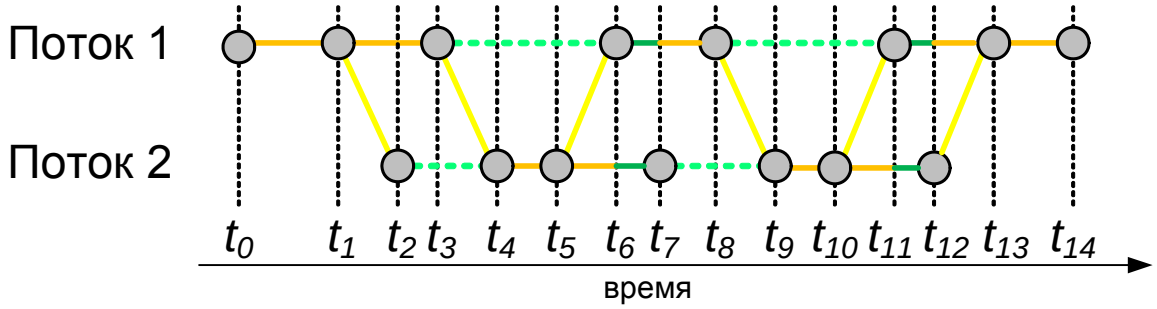

Рис. 4. Пример раскраски критического пути многопоточного приложения в соответствии с категориями времени

Итак, подведем итоги: профилирование многопоточного приложения основано на построении критического пути и его анализе с точки зрения эффективности использования процессора. Для этого выделяют категории времени, часть из которых считается проблемными и свидетельствует о неправильной организации многопоточного приложения. Деятельность разработчика, таким образом, состоит в нахождении проблемных участков критического пути, выявлении причин их появления (соотнесение с исходным кодом) и в последующем их разрешении.

4. Проблемы производительности, определяемые при помощи профилирования

Мы уже говорили ранее, что профилирование многопоточных приложений имеет свою специфику. ITP ориентирован на выявление проблемных ситуаций, характерных именно для многопоточных приложений, таких как: неэффективное управлением потоками, ошибки при выборе и использовании примитивов синхронизации, неправильное распределение вычислительной нагрузки и другие недостатки.

Рассмотрим далее наиболее распространенные ошибки и то, как ITP может помочь разработчикам в их обнаружении и исправлении.

4.1. Распределение вычислительной нагрузки

Весьма распространенной ошибкой является неравномерное распределение нагрузки между потоками. Общее время работы приложения в значительной степени зависит от времени работы самого медленного из его потоков. Очевидно, приложение будет работать быстрее всего, если нагрузка поделена между потоками поровну – тогда они завершают работу одновременно и за минимальное время.

Типичный пример: стоит задача обработки множества заявок, трудоемкость каждой из которых заранее неизвестна. В такой ситуации не всегда правильно разделять множество заявок поровну между потоками. Дело в том, что один поток может, например, получить все сложные заявки, в результате чего он станет узким местом в приложении.

Проблема распределения вычислительной нагрузки в ряде ситуаций решается достаточно просто. Первый признак ее появления – большая доля последовательных вычислений в приложении, что легко обнаруживается при анализе критического пути. Кроме того, ITP позволяет определить время работы каждого из потоков. При этом, если наблюдается существенное различие между временами работы нескольких потоков (при том, что они выполняют одинаковые действия), это свидетельствует о неравномерном распределении нагрузки.

Для рассмотренного выше примера в качестве решения можно предложить не делать априорного разделения множества заявок между потоками. Вместо этого стоит разрешить потокам выбирать из общей очереди новую заявку каждый раз после того, как была обработана предыдущая. Тогда пока один поток обрабатывает одну сложную заявку, второй поток успеет обработать несколько простых, и в результате можно ожидать более сбалансированного времени работы потоков.

4.2. Синхронизация и производительность

Аспекты производительности, связанные с синхронизацией между потоками, требуют особого внимания. Неправильно выбранная стратегия может привести к тому, что параллельная версия приложения будет выполняться даже медленнее, чем последовательная. Поэтому разработчик должен тщательно продумать используемую в его приложении модель синхронизации, и ITP способен оказать ему в этом существенную помощь.

Рассмотрим основные вопросы, связанные с синхронизацией.

4.2.1. Выбор примитивов синхронизации

Существует достаточно большое число примитивов синхронизации: мьютексы, семафоры, мониторы, критические секции. Все они имеют одинаковое назначение (обрамление критических областей программы), но эффективность (дополнительные накладные расходы) их работы может существенно различаться. При этом разработчику важно выбрать наиболее подходящий тип примитива синхронизации для каждой конкретной ситуации.

Рассмотрим, например, ситуацию, когда в приложении имеется некоторый целочисленный счетчик, являющийся глобальной переменной. При этом, если мы в приложении используем каждый раз увеличение этого счетчика на единицу, то выбор такого объекта как мьютекс является нерациональным. Гораздо эффективнее использовать атомарную функцию ОС Windows **`InterlockedIncrement`**. Кроме того, при разработке модели синхронизации следует делать выбор в пользу примитивов пользовательского уровня (**`CriticalSection`**, например), поскольку они работают быстрее, так как не генерируют системных вызовов операционной системы.

ITP способен помочь в выборе наиболее подходящего примитива, позволяя определить время, затраченное на работу с каждым из объектов синхронизации. Таким образом, разработчик может реализовать несколько моделей синхронизации и сравнить их производительность между собой.

4.2.2. Синхронизация между потоками

Разработчикам следует придерживаться следующей рекомендации: производить синхронизацию между потоками как можно реже. То есть необходимо сделать потоки максимально независимыми, чтобы избежать ситуаций, когда они ожидают друг друга. Слишком частое обращение нескольких потоков к разделяемому ресурсу приводит к тому, что большое количество времени потоки простаивают, находясь в состоянии ожидания.

Часто встречающаяся ситуация – это совместное использование одного и того же объекта несколькими потоками. Чтобы избежать блокировки потоков при обращении к объекту, часто используется подход, при котором каждый из потоков получает копию объекта в свое личное пользование. Так, в частности, поступают при работе нескольких пользователей с одной таблицей базы данных.

В данной ситуации ITP используется следующим образом. С его помощью определяются объекты, обращение к которым происходит наиболее часто. Затем анализируется насколько затратно столь частое обращение к объекту. Если обнаруживается, что производительность существенно страдает, разработчику следует попытаться изменить архитектуру приложения. Хороший пример – замена глобальных переменных локальными.

4.3. Непроизводительные издержки при работе с потоками

Важный момент при работе с потоками – появление дополнительных накладных расходов. Конечно, эти затраты гораздо меньше чем при управлении процессами, но все равно они могут оказаться весьма существенными. Основная рекомендация здесь состоит в следующем: потоки за время своей жизни должны совершать работу гораздо большей сложности, чем трудоемкость их собственного создания, уничтожения и управления ими.

Понятно, что если накладные расходы на управление потоками будут преобладать над их полезной деятельностью, то производительность приложения только пострадает.

ITP позволяет определить долю непроизводительных издержек от общего времени работы приложения. Если она оказывается слишком велика, скорее всего, приложение требует перепроектирования. Так, например, если создается система, обслуживающая поступающие в режиме реального времени запросы, то для обработки лучше содержать пул потоков, вместо того, чтобы создавать новый поток для каждого очередного запроса.

5. Общий порядок работы с инструментом

Порядок работы с Intel® Thread Profiler включает в себя следующие шаги:

- инструментация приложения;
- профилирование приложения;
- анализ полученной информации.

На первом шаге происходит подготовка приложения к профилированию – так называемая инструментация. Затем осуществляется запуск, и в процессе выполнения происходит накопление статистической информации о работе приложения. Собранная информация представляет собой трассу приложения, которая затем обрабатывается ITP и представляется в графическим виде, удобном для понимания. После этого разработчик может начинать непосредственно анализ производительности приложения.

Далее мы приведем рассмотрение процессов инструментации и профилирования. О том, как эти процессы осуществляются в ITP, мы расскажем в разделе 6.3.

5.1. Инструментация приложения

Инструментация представляет собой встраивание в приложение дополнительных вызовов, при помощи которых профилировщик получает информацию о работе приложения. Эти вызовы могут быть добавлены как на уровне исходного кода приложения, так и в уже скомпилированное приложение. В связи с этим различают *инструментацию исходных кодов* (*source instrumentation*) и *бинарную инструментацию* (*binary instrumentation*). ITP поддерживает оба этих способа.

Бинарная инструментация выполняется автоматически при запуске приложения из ITP. То есть вы имеете возможность взять любое уже скомпилированное приложение и немедленно начать профилировку. Однако полезность информации, которую вы получите, существенно зависит от опций, с которыми было скомпилировано приложение. Так, если в него была включена отладочная информация, то вы сможете обращаться к исходному коду из ITP. В противном случае вам будет доступен лишь ассемблерный код, что может быть весьма неудобно.

В связи с этим, обычная процедура состоит в следующем: разработчик компилирует свое приложение со всеми необходимыми опциями, а затем использует бинарную инструментацию. Мы рассмотрим этот процесс подробнее в разделе 6.2.

Второй тип инструментации используется крайне редко. Создатели ITP рекомендуют инструментацию исходных кодов лишь в двух ситуациях:

- Бинарная инструментация недоступна. Это справедливо для систем, созданных для работы на архитектурах Intel® Itanium и Intel® EM64T.
- Необходимо запустить инструментированное приложение вне Intel® Thread Profiler.

Далее под инструментацией мы будем понимать именно бинарную инструментацию, поскольку именно с ней нам придется работать.

5.2. Профилирование приложения

После того как приложение было инструментировано, можно начинать профилирование. ITP осуществляет запуск приложения, в процессе которого собирает его трассу. В нее включается следующая статистическая информация:

- идентификаторы созданных потоков;
- время создания и уничтожения каждого потока;
- количество времени, которое каждый поток провел в состоянии ожидания;
- эффективность использования приложением ядер процессора на каждом участке критического пути.

ITP также регистрирует большое число других событий и вызовов API, информация о которых может быть полезна при оптимизации производительности многопоточного приложения.

При профилировании старайтесь следовать следующим советам:

- Избегайте запускать другие приложения во время профилирования. Деятельность посторонних приложений (особенно потребляющих много ресурсов) может существенно исказить интересующую вас информацию.
- Производите профилирование несколько раз и для анализа выбирайте тот запуск, когда приложение отработало быстрее всего. Этот запуск соответствует ситуации, когда на ваше приложение было

меньше всего воздействий, поэтому эта информация ближе всего к «идеальному» профилю вашего приложения.

Далее мы на конкретном примере познакомимся с графическим интерфейсом ITP и основными приемами работы с ним.

6. *Пример использования Intel Thread Profiler*

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

Настоящий раздел представлен в виде лабораторной работы, которую читатели могут выполнять одновременно с чтением данного документа.

6.1. 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 факторизуемого 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, факторизуемое число делится на него, и алгоритм завершает работу, когда частное от очередного деления становится равным единице. 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 **C:\ITPLab\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** (рис.5). После этого в рабочей области **Microsoft Visual Studio** появится программный код, с которым нам предстоит работать.

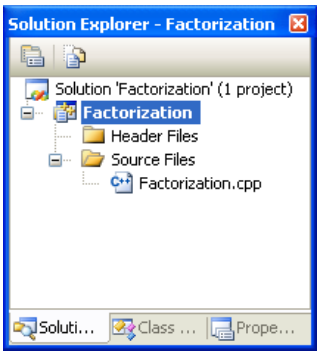

Рис. 5. Открытие файла Factorization.cpp

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.

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 рис. 6

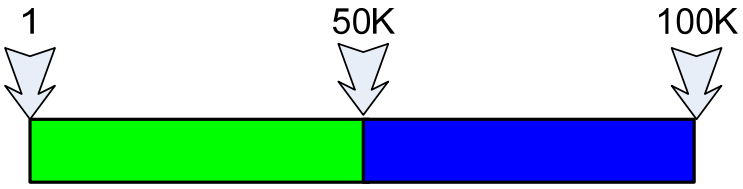

Рис. 6. Распределение нагрузки между потоками

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.

6.2. Подготовка приложения для профилирования

Для того чтобы приложение можно было профилировать при помощи ITP, 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**.

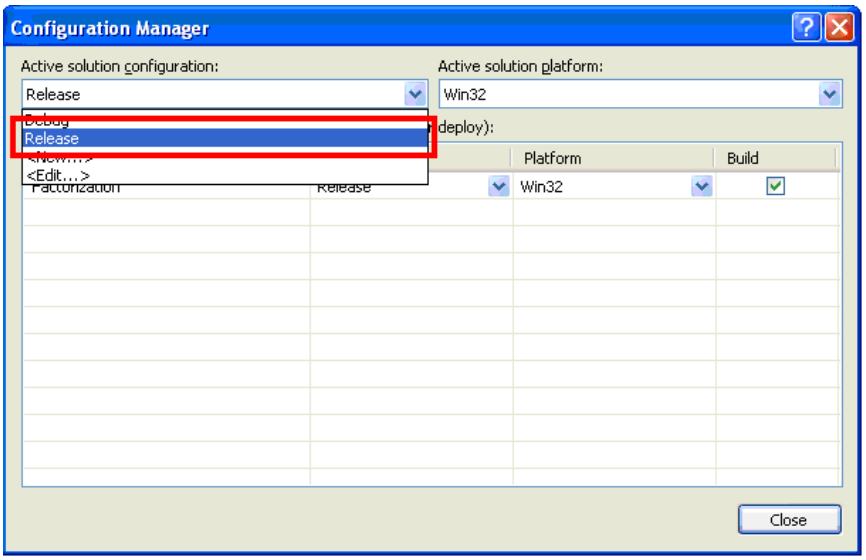

Рис. 7. Выбор режима Release

2. В меню **Project** выберите пункт **Properties**, в результате чего появится окно, представленное на рис. 7. 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**.

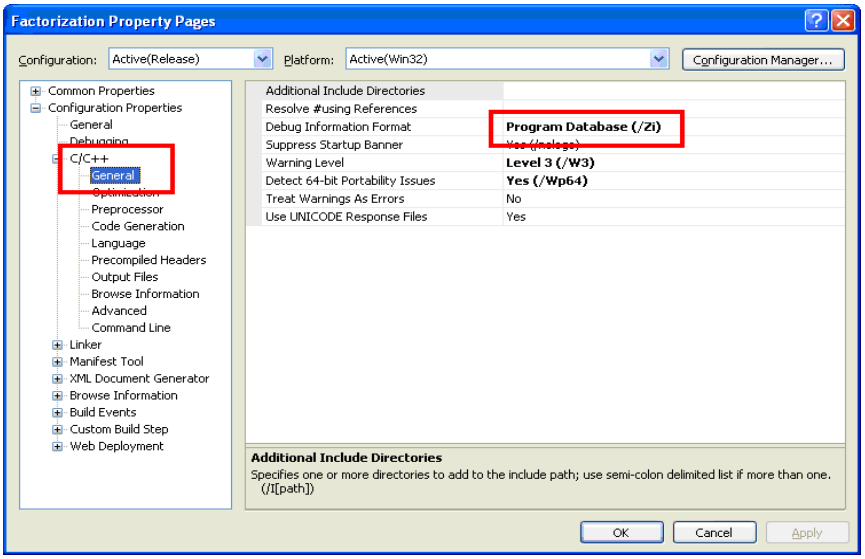

Рис. 8. Указание формата отладочной информации

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**.

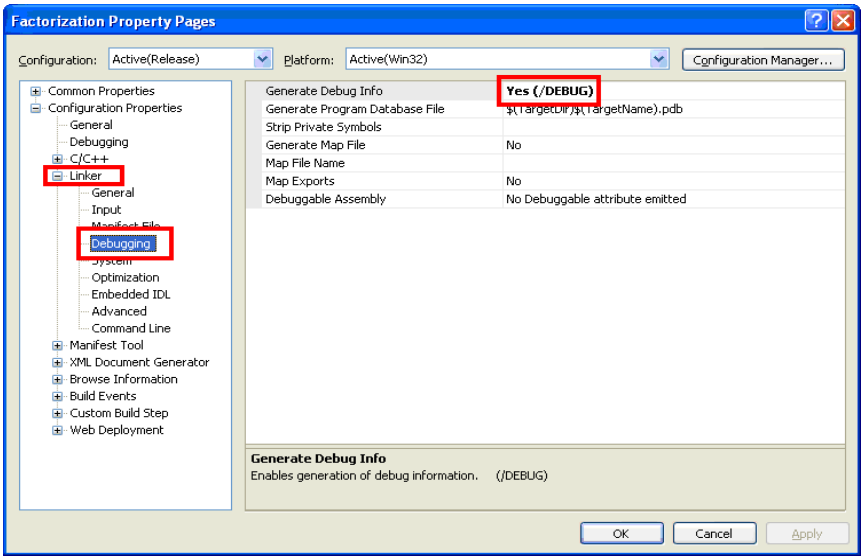

Рис. 9. Указание генерации отладочной информации

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**.

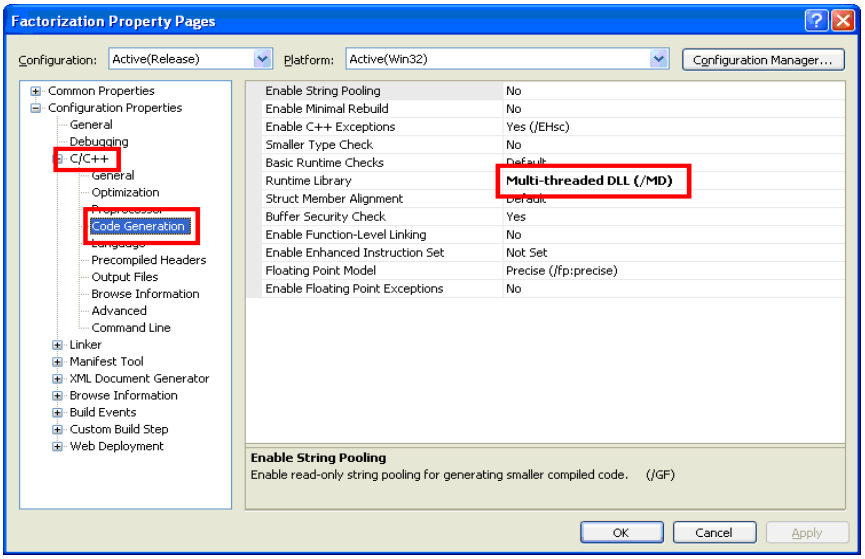

Рис. 10. Выбор потокобезопасных библиотек

5. Убедитесь, что приложение компонуется с использование 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**.

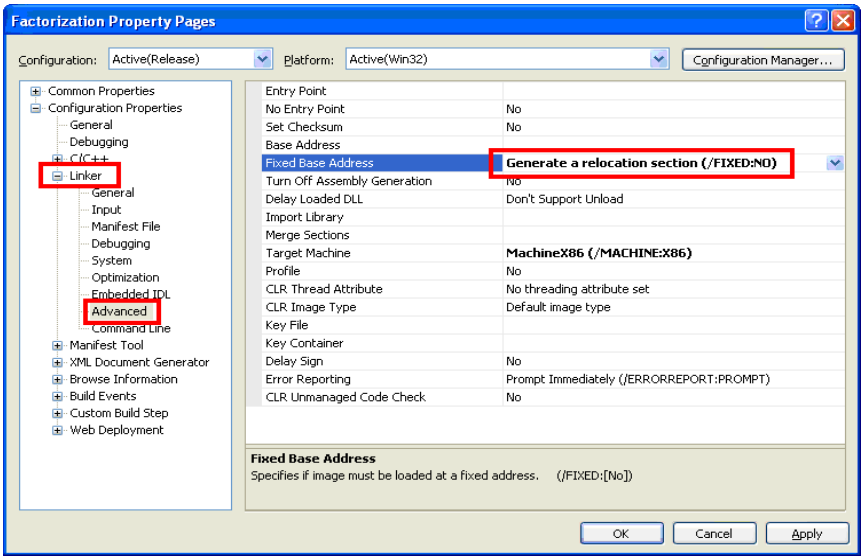

Рис. 11. Установка опций компоновщика

После выполнения данных процедур ваше приложение готово к профилированию.

6.3. Профилирование приложения

6.3.1. 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 3.0→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**.

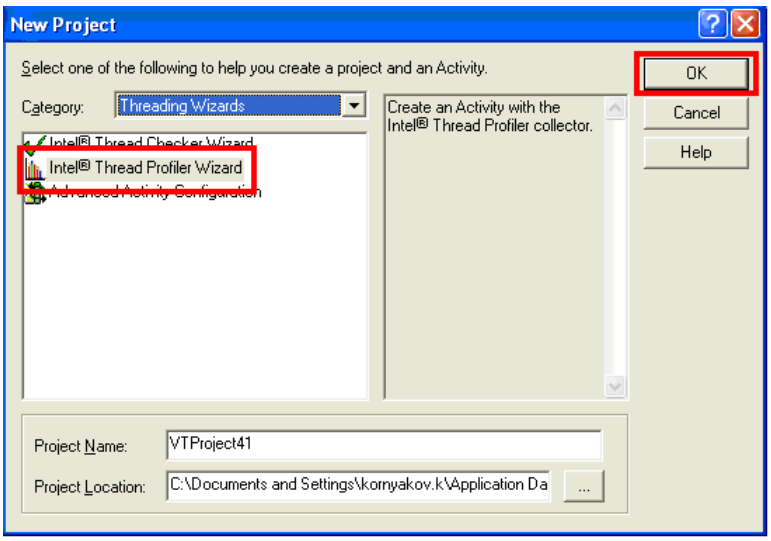

Рис. 12. Выбор типа проекта

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 **C:\ITPLab\Factorization\release\Factorization.exe**.

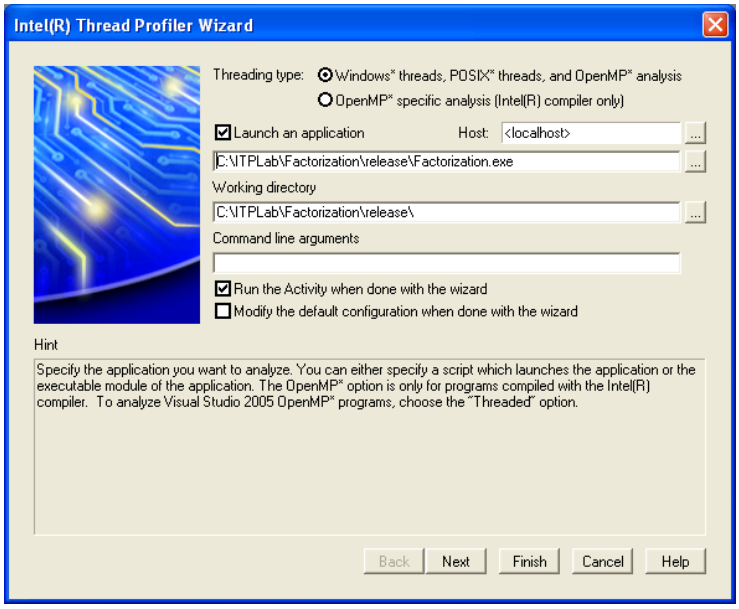

Рис. 13. Выбор профилируемого приложения

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** (рис. 14). Последние два окна являются основными источниками информации о критическом пути приложения.

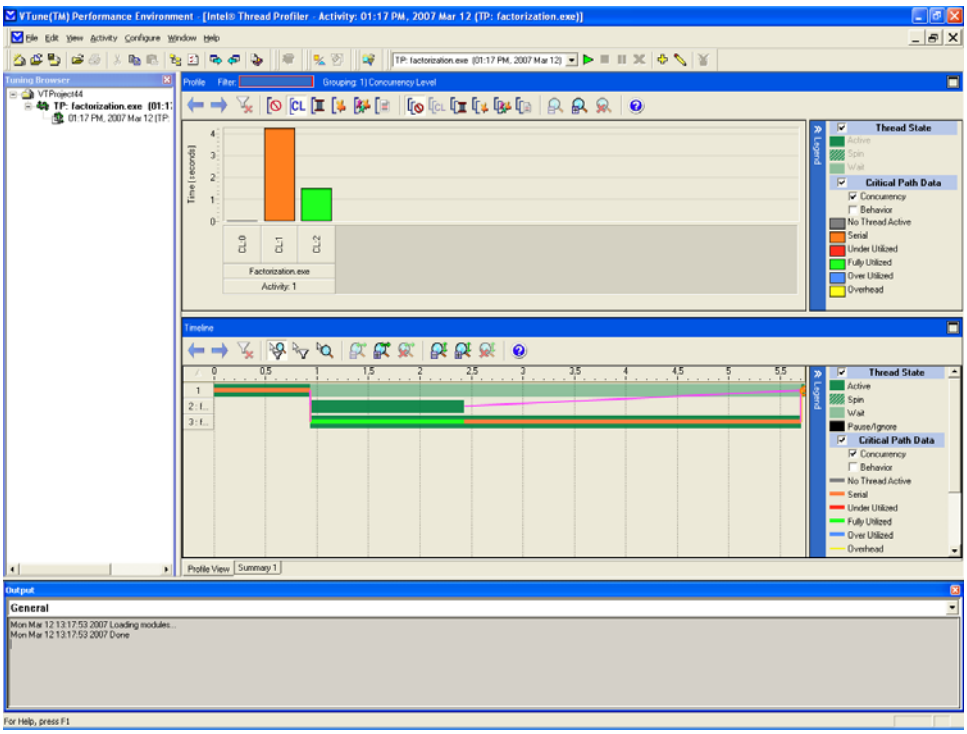

Рис. 14. Рабочая область Intel Thread Profiler

6.3.2. 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 пункт меню **View→Tuning Browser**.

6.4. Анализ производительности приложения

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

6.4.1. Вкладка Summary

В нижней части окна ITP 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 (рис. 15).

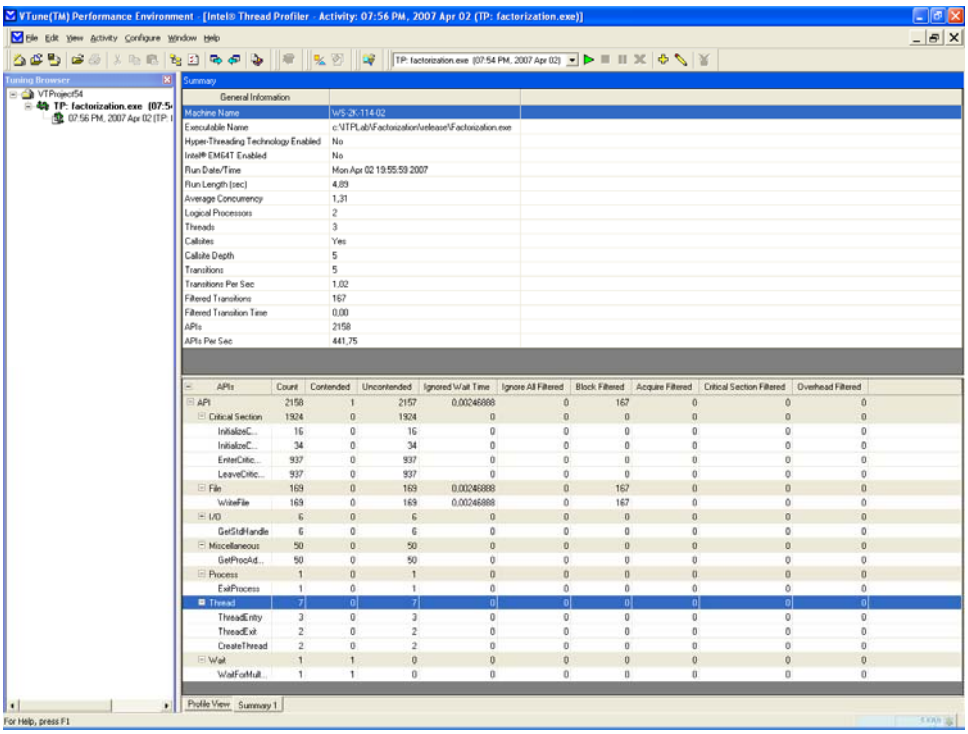

Рис. 15. Вкладка Summary

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 знак "+", располагающийся возле надписи “APIs” в нижней таблице, при этом она должна принять вид, как показано на рис. 15. 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**.

6.4.2. 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 справка по использованию цветов (*легенда*), располагающаяся справа. Рассмотрим их подробнее.

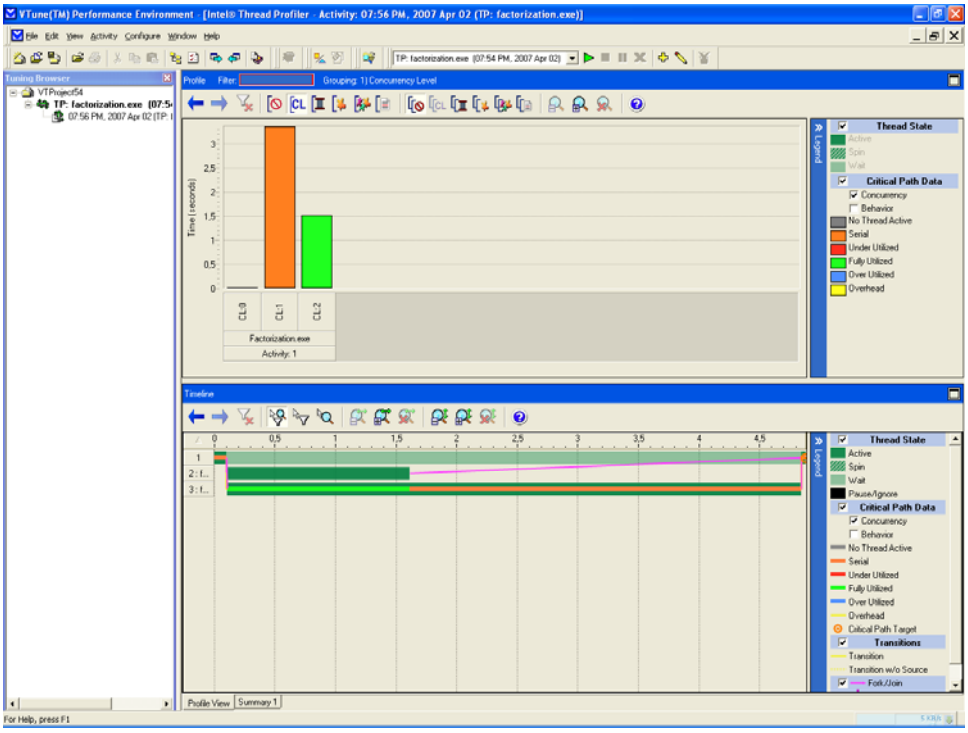

Рис. 16. Окно Profile

6.4.2.1. 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 рис. 17.

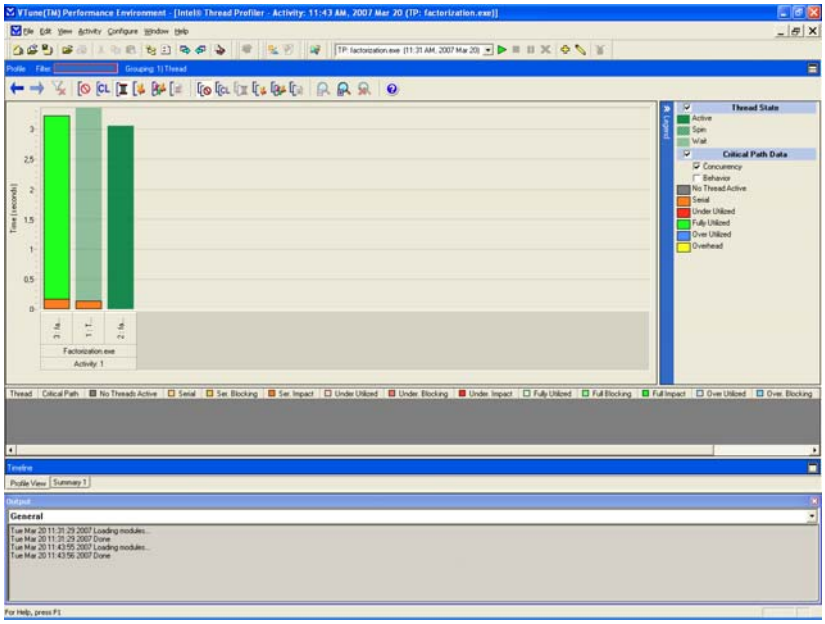

Рис. 17. Анализ состояний потоков

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*). Здесь мы имеем дело с категориями времени, про которые было рассказано в разделе 3. 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, о котором мы говорили ранее в разделе 3. 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.

6.4.2.2. 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 (кнопка ), то можно проанализировать эффективность работы каждого из потоков. Если же включить группировку по объектам (кнопка ), 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 рис. 18.

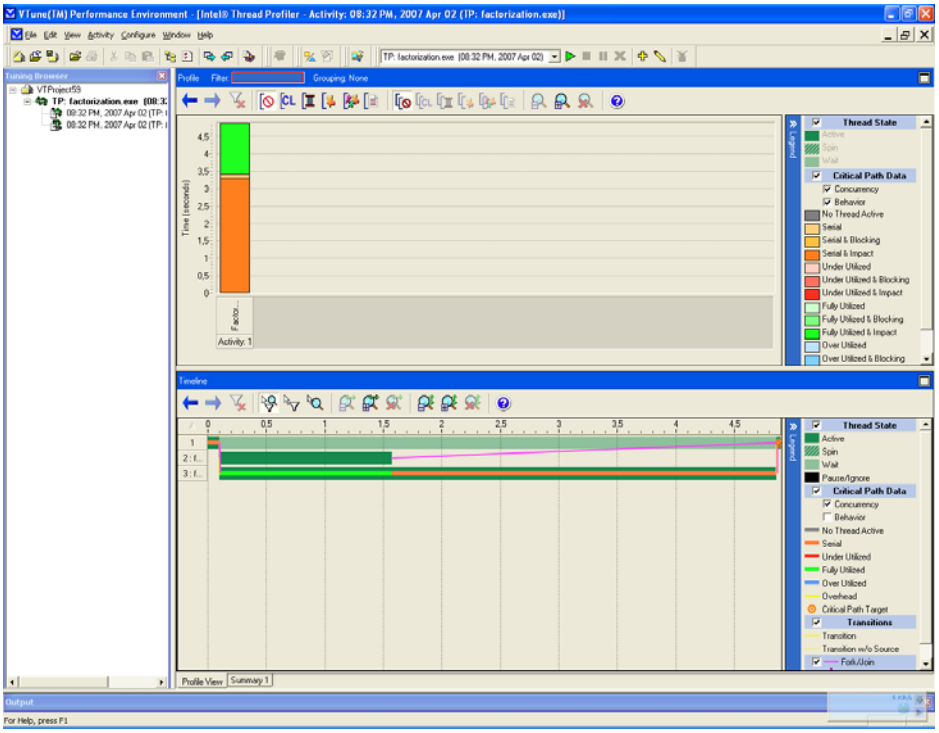

Рис. 18. Изображение критического пути при отключенной группировке

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 рис. 19.

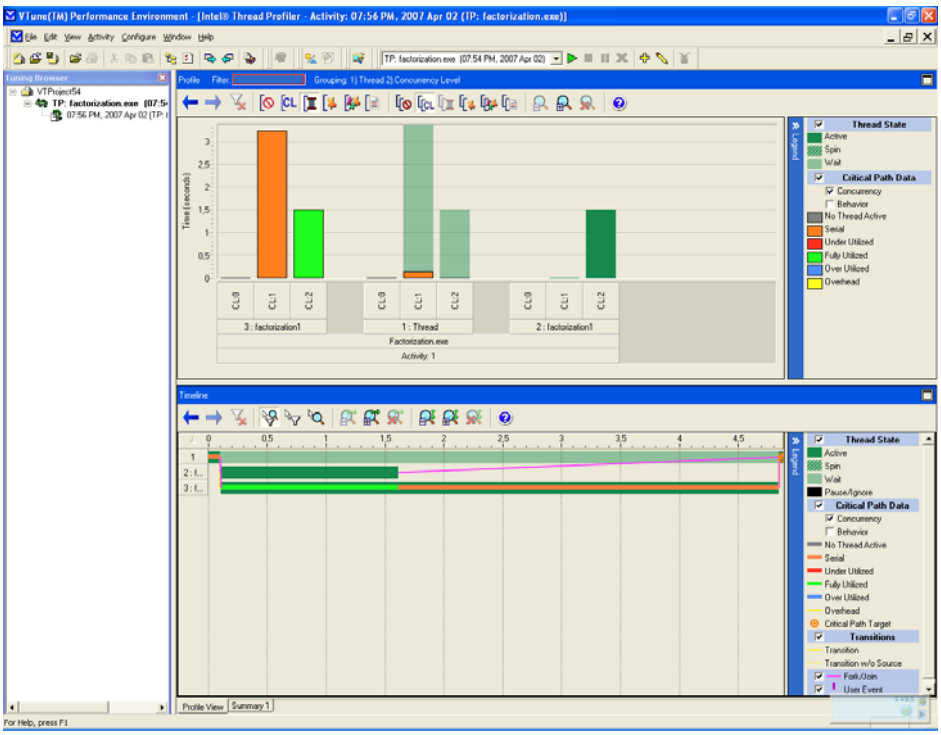

Рис. 19. Использование вторичной группировки

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, то есть параллельно с ним не работал ни один другой поток. Второй же поток (правая группа столбиков) практически все время работал параллельно с третьим потоком. Основной 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.

6.4.2.3. 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 рис. 20. 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.

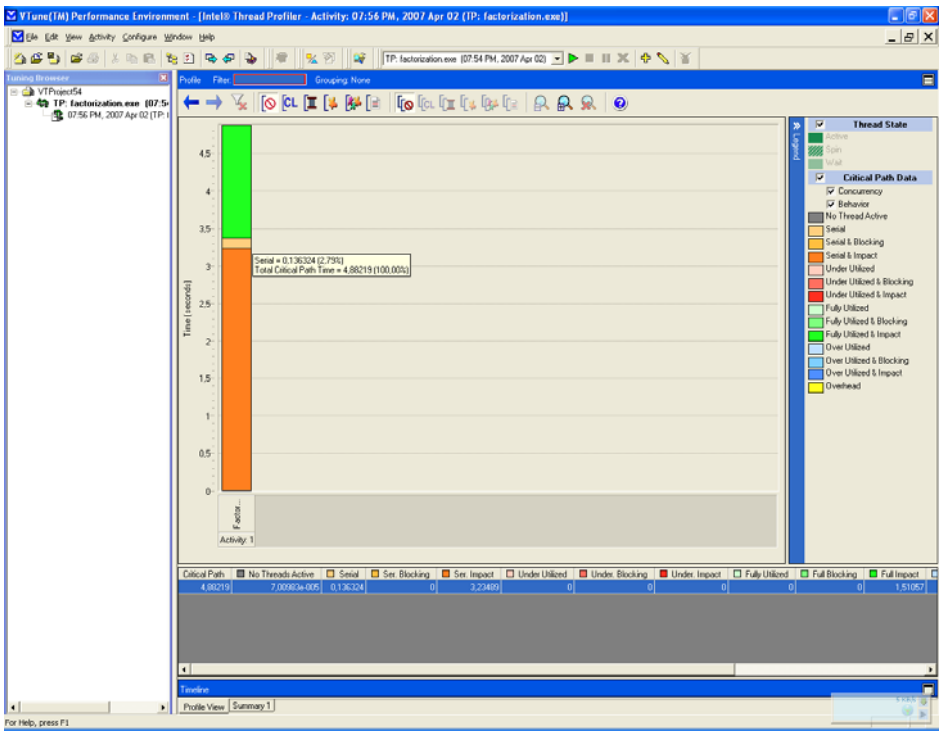

Рис. 20. Анализ критического пути

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 рис. 21.

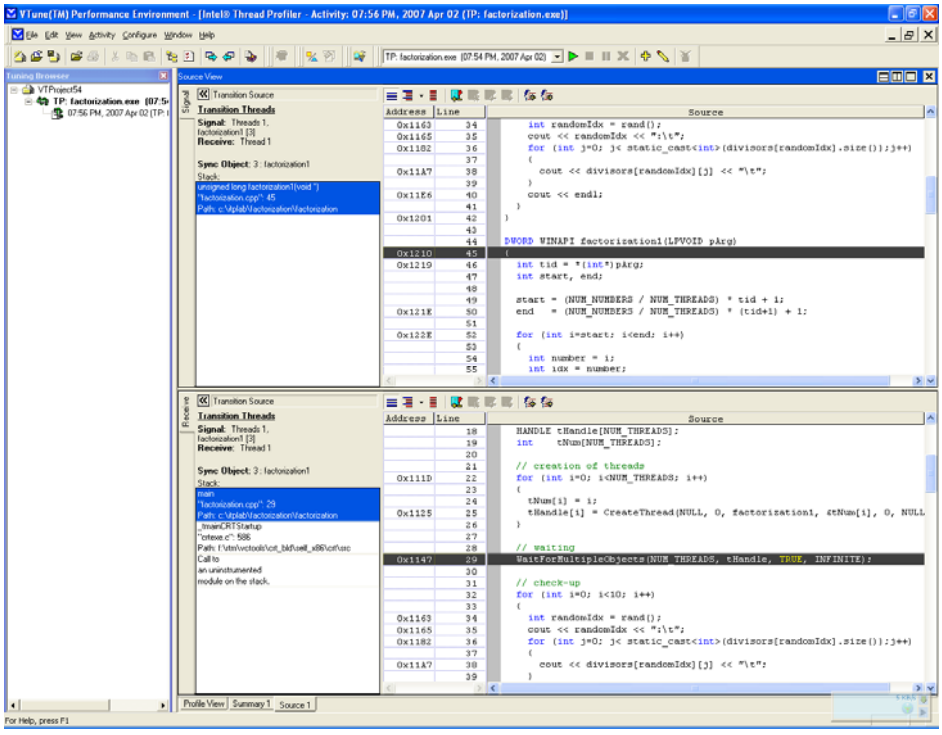

Рис. 21. Обращение к исходному коду приложения

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 .

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 ITP 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 ITP – окну **Timeline**.

6.4.3. 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.

6.4.3.1. 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.

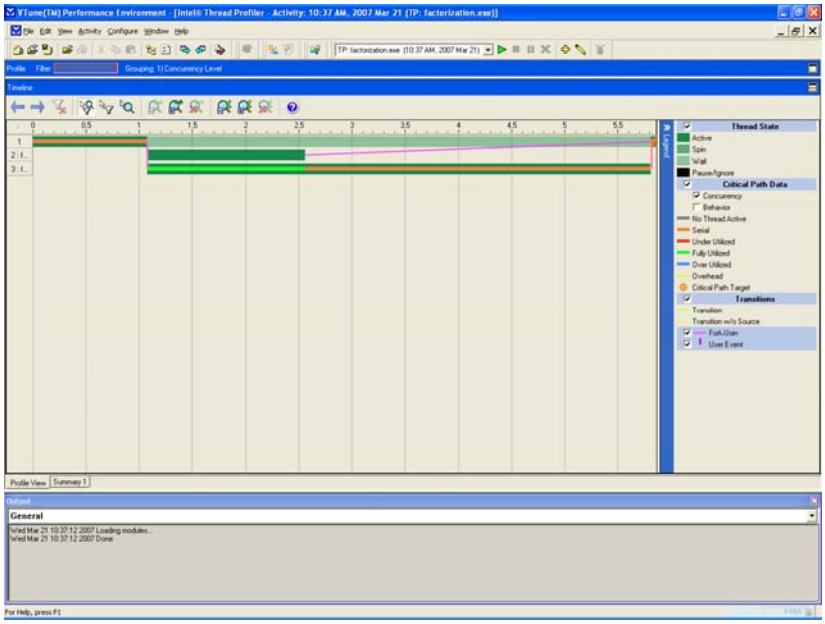

Рис. 22. Окно Timeline

На рис. 22 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, зажмите

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 рис. 23.

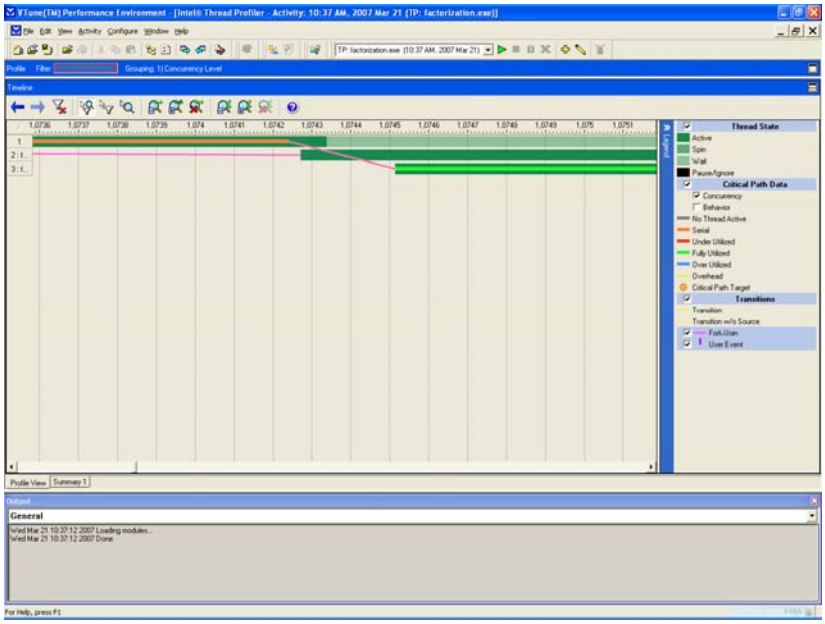

Рис. 23. Момент создания потоков

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 рис. 21.

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 (рис. 24).

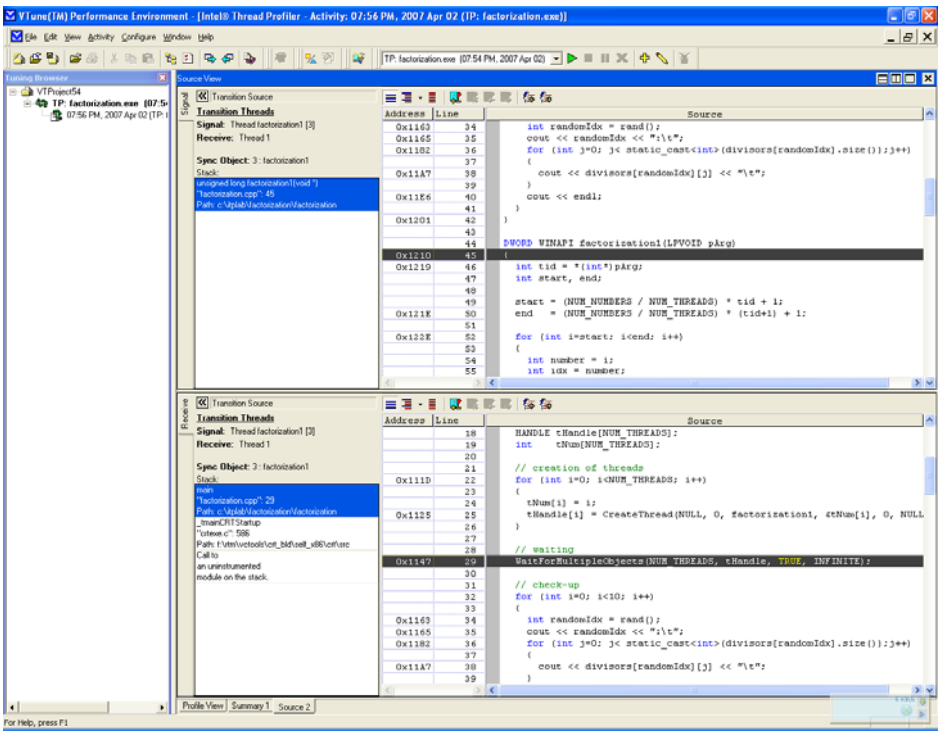

Рис. 24. Место ожидания основного потока

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

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

6.4.3.2. 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 лабораторной работе 1.

6.5. 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. Как было установлено, основная причина медленной работы учебного приложения – неравномерное распределение нагрузки между потоками. Предложите ваш способ решения данной проблемы.

7. 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».