Лабораторная работа
Балансировка нагрузки в многопоточной программе с использованием Intel Thread Profiler

Содержание

1 Цель лабораторной работы

Изучить основные вопросы, возникающие при балансировке вычислительной нагрузки между потоками. Рассмотреть различные подходы к балансировке, приобрести навыки анализа их реализаций и сравнения между собой при помощи Intel® Thread Profiler (ITP).

2 Инструкция для выполнения лабораторной работы

Предполагается, что в процессе начального ознакомления с ITP вы уже изучили приложение **Factorization**, производящее разложение множества чисел на простые множители (факторизация).

Подробное описание приложения может быть найдено в [8].

Как мы установили ранее, основная причина низкой производительности приложения **Factorization** заключается в неравномерном распределении нагрузки между потоками, производящими факторизацию.

Далее последовательно рассмотрим несколько различных подходов к балансировке и попытаемся определить, какой из них является наиболее эффективным для нашего приложения.

Замечание: результаты, которые будут получены вами, могут несколько отличаться от приведенных в настоящем документе. Повлиять на это может несколько факторов, в том числе характеристики вычислительного узла, на котором выполняется лабораторная работа, прежде всего количество ядер. Мы производили профилирование на двухъядерной машине.

2.1 Подход 1: разделение множества чисел на одинаковые части по числу потоков

Этот подход уже был рассмотрен ранее [8]. Однако, обратимся к нему еще раз, чтобы понять, в чем причины дисбаланса нагрузки между потоками. Одновременно с этим попытаемся найти более эффективный способ распределения множества чисел между потоками.

Откройте файл **Factorization.cpp** в проекте **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**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **Factorization.cpp**. Сейчас нас интересует рабочая функция потока под названием **factorization1**. Она реализует стратегию распределения нагрузки, которая

схематично представлена на рисунке 1. Первый поток получает множество чисел от 1 до 50000, а второй от 50001 до 100000.

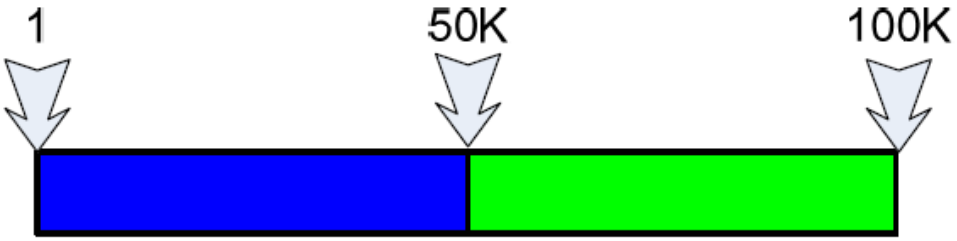

Рисунок 1 - Разделение множества чисел на одинаковые части по числу потоков

Убедитесь, что в программе в качестве аргументов функции **CreateThread** передается именно функция **factorization1**. В дальнейшем, чтобы использовать другой подход к распределению нагрузки, необходимо будет лишь заменить имя рабочей функции потока на **factorization2** или **factorization3**.

Далее необходимо подготовить приложение к профилированию, установив определенные опции компиляции и компоновки. Сделайте это, как указано в описании [8], после чего создайте проект в ITP и запустите процесс профилирования. Рабочая область ITP должна принять вид, как показано на рисунке 2.

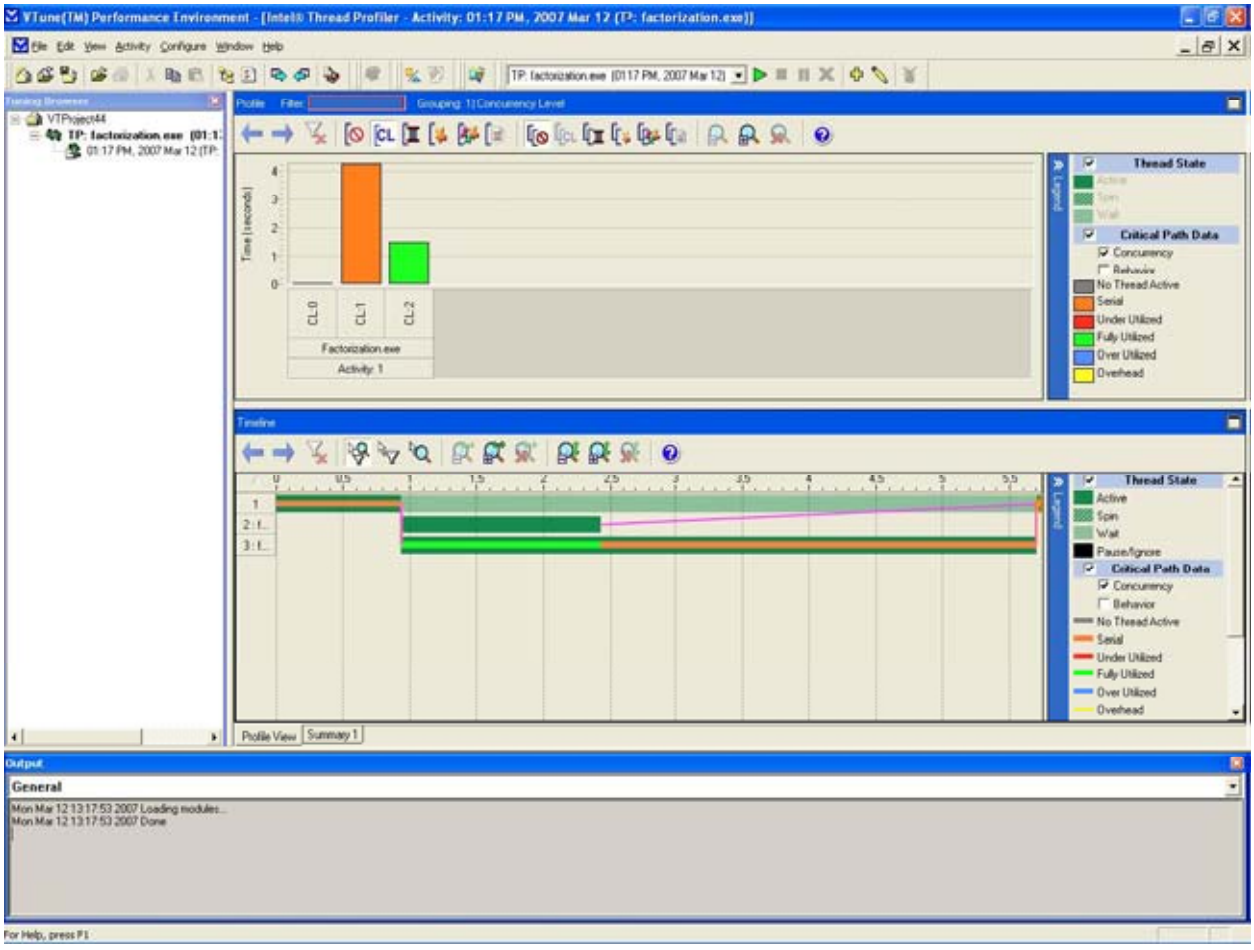

Рисунок 2 - Результат профилирования приложения, использующего первый подход к распределению нагрузки

В окне **Profile** можно заметить, что столбик оранжевого цвета оказался выше других. Это означает, что большую часть времени приложение выполнялось в последовательном режиме, то есть работал лишь один поток из трех. Далее обратите внимание на окно **Timeline**. После взгляда на него, становится ясно, в чем причина того, что в нашем приложении преобладает последовательное исполнение. Неравномерное распределение нагрузки привело к тому, что второй дочерний поток работает гораздо дольше первого и тормозит работу приложения в целом.

Обратите внимание на информацию, представленную в окне Summary. Зафиксируйте время работы приложения.

2.2. Подход 2: разделение множества чисел на четные и нечетные

Рассмотрим следующий способ распределения нагрузки. В данном случае первому потоку достанутся все нечетные числа, а второму все четные. На рисунке 3 синим цветом помечены числа, которые будет обрабатывать первый поток, а зеленым – второй.

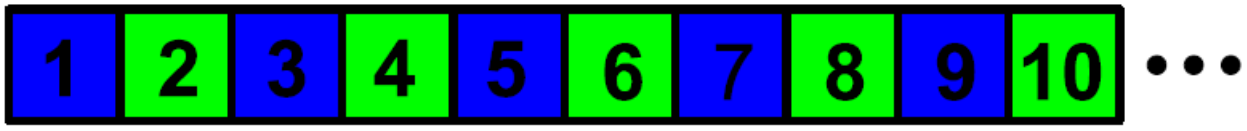

Рисунок 3 - Разделение множества чисел на четные и нечетные

Описанный подход реализован в функции **factorization2**. Ознакомьтесь с ней, и в месте вызова функции **CreateThread** замените ее аргумент **factorization1** на **factorization2**, чтобы приложение стало использовать разбиение множества чисел на четные и нечетные.

Перекомпилируйте приложение и запустите процесс профилирования. Сделайте это в рамках того же проекта в ITP, чтобы в окне **Tuning Browser** сохранились результаты предыдущего запуска, и мы могли впоследствии сравнить их с новыми результатами. Рабочая область ITP должна принять вид, как показано на рисунке 4.

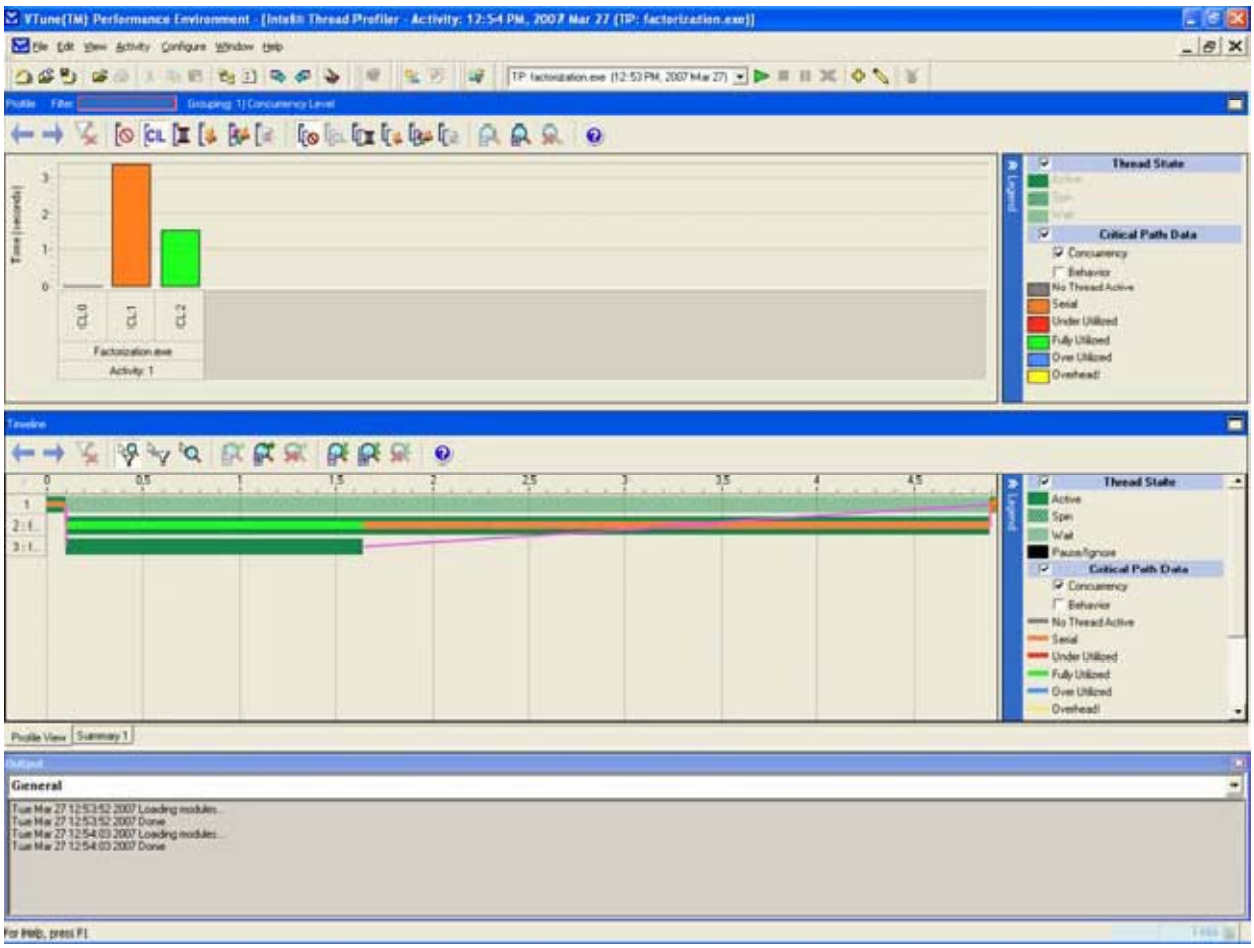

Рисунок 4 - Результат профилирования приложения, использующего второй подход к распределению нагрузки

Результаты окажутся неожиданными на первый взгляд. Высота оранжевого столбика практически не изменилась, но потоки как будто поменялись местами. Теперь первый из дочерних потоков выполняется гораздо медленнее второго.

Зафиксируйте время работы приложения.

2.3. Подход 3: разделение множества чисел на небольшие пачки

Итак, мы уже рассмотрели два подхода к балансировке нагрузки, но они оказались неэффективными из-за специфики нашей задачи. Нам нужен способ, позволяющий более равномерно нагрузить потоки.

Существует довольно популярный метод в задачах обработки множества однотипных заявок – обработка их целыми пачками. Все множество заявок разделяется на множество пачек некоторой длины, после чего эти пачки распределяются поровну между потоками. Это изображено схематично на рисунке 5 (размер пачки равен 1000).

Рисунок 5 - Разделение множества чисел на пачки

Реализован этот подход в функции **factorization3**. Ознакомьтесь с ее кодом и перекомпилируйте приложение, подставив **factorization3** в качестве аргумента функции **CreateThread**. Перейдите обратно в ITP и запустите процесс профилирования. При этом должны появиться графики, представленные на рисунке 6.

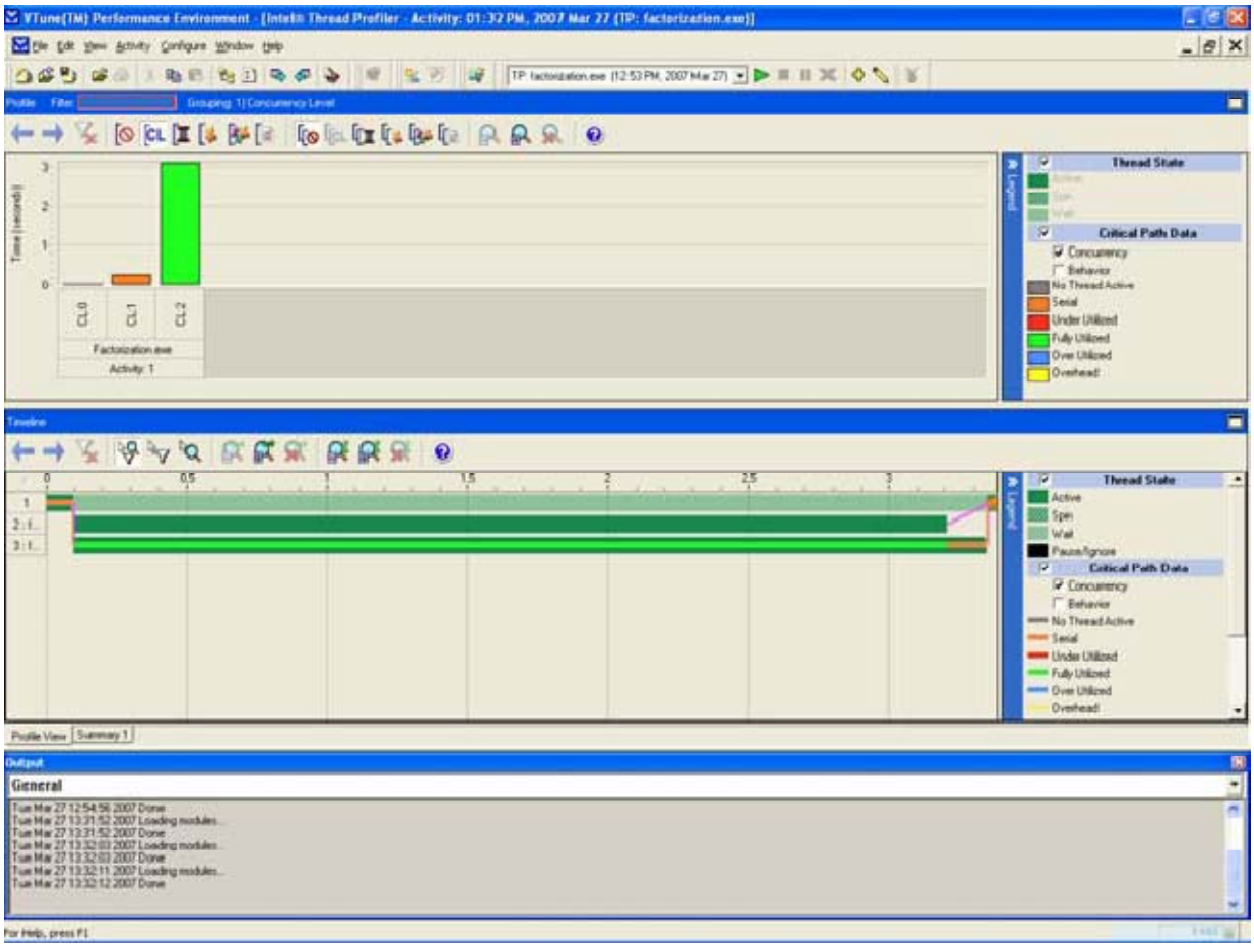

Рисунок 6 - Результат профилирования приложения, использующего третий подход к распределению нагрузки

В этот раз мы достигли успеха. Основную часть времени приложение выполняется в параллельном режиме, а время работы существенно сократилось, что означает увеличение производительности приблизительно на 30%!

Итак, нам все-таки удалось подобрать верную стратегию распределения нагрузки между потоками.

Зафиксируйте время работы приложения.

Резюме

- Нередко причина низкой производительности – большая доля последовательных вычислений в приложении.
- ITP позволяет определить участки последовательного исполнения и время работы каждого из потоков.
- Далее разработчик либо распараллеливает последовательный участок, либо перераспределяет нагрузку.

3 Самостоятельная работа

3.1 Выбор оптимальной степени гранулярности

Под *гранулярностью* понимают детальность разбиения исходной задачи на подзадачи. В нашем случае степень гранулярности означает размер пачек чисел, которые мы используем в третьем подходе распределения нагрузки. За этот параметр отвечает константа **GRAIN_SIZE**, которая объявлена непосредственно перед кодом функции **factorization3**.

Определение оптимальной степени гранулярности – самостоятельная задача в некоторых приложениях.

В нашем случае, если размер пачки очень велик (например, 50000), то подход выродится в первый из рассмотренных нами. Слишком сильное дробление исходной задачи тоже не эффективно, особенно если требуется синхронизация между потоками после обработки очередной пачки (сохранение промежуточных результатов в глобальные переменные). В некоторых случаях время на синхронизацию может даже превышать выигрыш от параллельной обработки, поэтому степень гранулярности нужно всякий раз подбирать очень аккуратно. В нашем приложении синхронизации потоков не производится, но явление снижения производительности при малом размере пачки все равно имеет место.

Путем экспериментов определите оптимальную степень гранулярности. Попытайтесь найти объяснение, почему именно найденная вами величина дает максимальную производительность.

3.2 Контрольные вопросы

1 Сравните время работы приложения при использовании разных подходов. Назовите причины, приводящие к полученным результатам в каждом из случаев.
2 В задачах какого класса должен хорошо показать себя первый из описанных подходов распределения нагрузки?
3 Как вы думаете, если использовать второй подход с четырьмя потоками, равномерно ли распределится нагрузка между ними? Почему? Проведите эксперимент и проверьте свои предположения.
4 Предложите свой метод распределения нагрузки между потоками. Если имеется время, запрограммируйте его и сравните с подходами, приведенными в настоящем документе.
5 Как вы думаете, какой должен быть следующий шаг, если мы хотим продолжить повышение производительности приложения **Factorization**?

4 Содержание отчета

Отчет должен содержать:

1 Результаты самостоятельной работы по выбору оптимальной степени гранулярности при использовании третьего подхода (раздел 3.1).

2 Ответы на контрольные вопросы.

Отчёт необходимо сопроводить соответствующей пояснительной информацией (численные значения, снимки экрана, примеры кода).

5 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».

8 «Intel® Thread Profiler. Краткое описание».