

Лабораторная работа №1: балансировка нагрузки

Содержание

1.	Цель лабораторной работы	1
2.	Инструкция для выполнения лабораторной работы.....	1
2.1.	Подход 1: разделение множества чисел на одинаковые части по числу потоков.....	1
2.2.	Подход 2: разделение множества чисел на четные и нечетные.....	2
2.3.	Подход 3: разделение множества чисел на небольшие пачки.....	3
3.	Самостоятельная работа	4
3.1.	Выбор оптимальной степени гранулярности.....	4
3.2.	Контрольные вопросы.....	4
4.	Литература.....	5

1. Цель лабораторной работы

Изучить основные вопросы, возникающие при распределении нагрузки между потоками. Рассмотреть различные подходы к балансировке, приобрести навыки их анализа 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 мы 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 могут 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 нами ранее. Мы, 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 **C:\ITPLab\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 рис. 1. Первый поток получает множество чисел от 1 до 50000, а второй от 50001 до 100000.

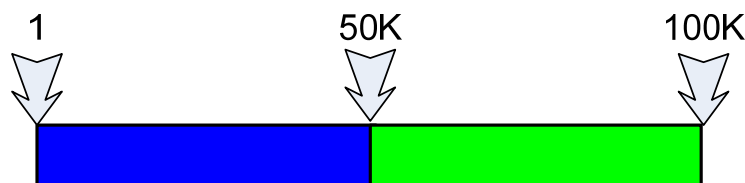


Рис. 1. Разделение множества чисел на одинаковые части по числу потоков

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. Сделайте это, как указано в описании [8], после чего создайте проект в ИТР и запустите процесс профилирования. Рабочая область ИТР должна принять вид, как показано на рис.2.

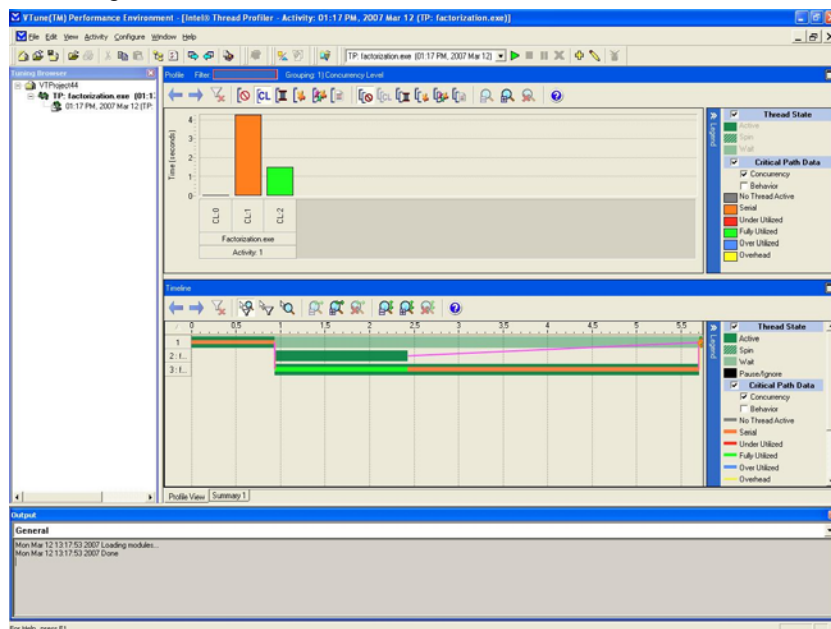


Рис. 2. Результат профилирования приложения, использующего первый подход к распределению нагрузки

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 преобладания последовательного исполнения. Неравномерное распределение нагрузки привело к тому, что второй дочерний поток работает гораздо дольше первого и тормозит работу приложения в целом.

Недостаток выбранного подхода распределения нагрузки очевиден: первый поток факторизует числа от 1 до 50000, что существенно проще, чем разложить числа от 50000 до 100000. Итак, наша задача состоит в том, чтобы придумать более эффективный способ разделения чисел между потоками.

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

Рассмотрим следующий способ распределения нагрузки. Если недостаток предыдущего подхода состоит в том, что потоки обрабатывали различные по величине числа, то нужно исправить это. Проще всего сделать так, чтобы они брали числа по очереди. То есть в нашем случае первому потоку достанутся все нечетные числа, а второму все четные. На рис. 3 синим цветом помечены числа, которые будет обрабатывать первый поток, а зеленым – второй.



Рис. 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 ИТР, чтобы в окне **Tuning Browser** сохранились результаты предыдущего запуска, и мы могли впоследствии сравнить их с новыми результатами. Рабочая область ИТР должна принять вид, как показано на рис. 4.

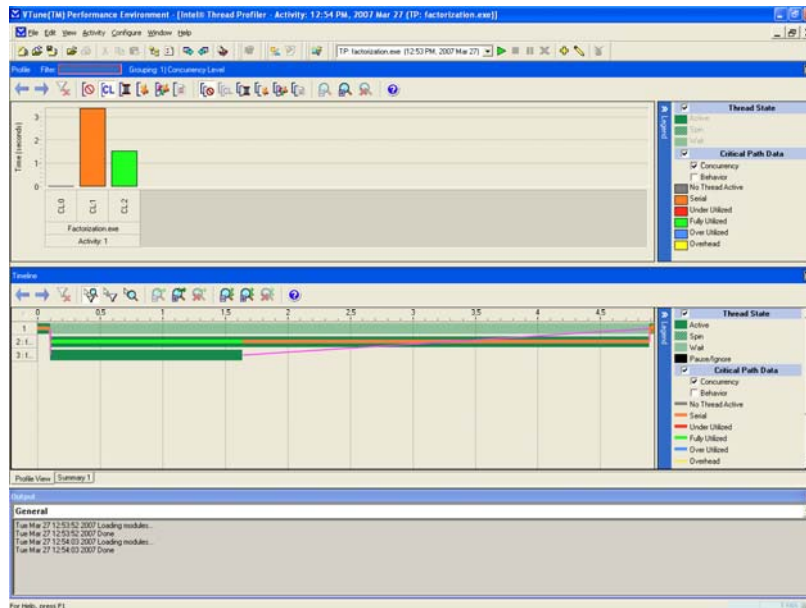


Рис. 4. Результат профилирования приложения, использующего второй подход к распределению нагрузки

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.

Причина и в этот раз находится на поверхности. Четные числа факторизовать существенно проще, чем нечетные, поскольку после первого же деления (на число 2) их величина сокращается в два раза. Мы снова сталкиваемся с ситуацией, когда один поток вынужден совершать работу гораздо более трудоемкую, чем второй.

Очевидно, и этот подход дает неудовлетворительные результаты.

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 небольшой длины, после чего эти пачки распределяются поровну между потоками. Это изображено схематично на рис. 5 (размер пачки равен 1000).

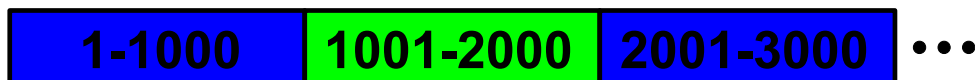


Рис. 5. Разделение множества чисел на небольшие пачки

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

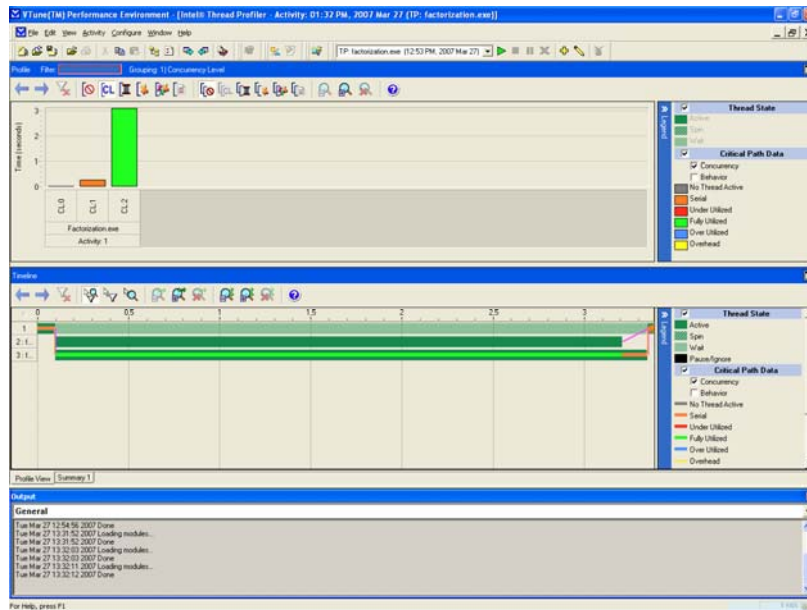


Рис. 6. Результат профилирования приложения, использующего третий подход к распределению нагрузки

В этот раз мы достигли успеха. Основную часть времени приложение выполняется в параллельном режиме, а время работы сократилось с 5 до 3,5 секунд, что означает увеличение производительности на 30%!

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

3. Самостоятельная работа

3.1. Выбор оптимальной степени гранулярности

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

Определение оптимальной степени гранулярности – самостоятельная задача в некоторых приложениях. В нашем случае, если размер пачки очень велик (например, 50000), то подход вырождается в первый из рассмотренных нами. Слишком мелкое 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 выбирать очень осторожно. В нашем приложении синхронизации не производится, но явление снижения производительности при малом размере пачки все равно имеет место.

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

3.2. Контрольные вопросы

- В задачах какого класса должен хорошо показать себя первый из описанных подходов распределения нагрузки?
- Как вы думаете, если использовать второй подход с четырьмя потоками, равномерно ли распределится нагрузка между ними? Почему? Проведите эксперимент и проверьте свои предположения.
- Предложите свой метод распределения нагрузки между потоками. Если имеется время, запрограммируйте его и сравните с подходами, приведенными в настоящем документе.
- Как вы думаете, какой должен быть следующий шаг, если мы хотим продолжить повышение производительности приложения **Factorization**?

4. 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», материалы по образовательному комплексу «Технологии разработки параллельных программ».