Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс «Параллельные численные методы»

Лабораторная работа №1
Вычисление простых чисел

Кустикова В.Д., Сиднев А.А., Сысоев А.В.

При поддержке компании Intel

Нижний Новгород
2010

Содержание

ВВЕДЕНИЕ **4**

БЛАГОДАРНОСТИ **4**

1. МЕТОДИЧЕСКИЕ УКАЗАНИЯ **4**

1.1. Цели и задачи работы 4
1.2. Структура работы 5
1.3. Тестовая инфраструктура 5

2. ИНСТРУМЕНТ INTEL PARALLEL STUDIO **6**

3. ЗАДАЧА РАЗЛОЖЕНИЯ ЧИСЕЛ НА ПРОСТЫЕ МНОЖИТЕЛИ **7**

4. РАЗДЕЛЕНИЕ МНОЖЕСТВА ЧИСЕЛ НА ОДИНАКОВЫЕ ЧАСТИ ПО КОЛИЧЕСТВУ ПОТОКОВ **8**

4.1. Задание 1. Открытие проекта 01_EqualPartition 8
4.2. Задание 2. Реализация параллельного алгоритма. Поиск ошибок работы с памятью и ошибок многопоточности 10
4.3. Задание 3. Анализ эффективности («горячие» точки) 14
4.4. Задание 4. Оценка степени параллелизма 16

5. РАЗДЕЛЕНИЕ МНОЖЕСТВА ЧИСЕЛ С ЧЕРЕДОВАНИЕМ **17**

5.1. Задание 1. Открытие проекта 02_EvenOdd 17
5.2. Задание 2. Реализация параллельного алгоритма 18
5.3. Задание 3. Оценка степени параллелизма 18

6. РАЗДЕЛЕНИЕ МНОЖЕСТВА ЧИСЕЛ НА ГРУППЫ **20**

6.1. Задание 1. Открытие проекта 03_Blocks 20
6.2. Задание 2. Реализация параллельного алгоритма 20
6.3. Задание 3. Оценка степени параллелизма 21

7. КОНТРОЛЬНЫЕ ВОПРОСЫ **22**

8. ДОПОЛНИТЕЛЬНЫЕ ЗАДАНИЯ **22**

9. ЛИТЕРАТУРА **23**

ПРИЛОЖЕНИЕ 1. ПРОГРАММНЫЙ КОД ОСНОВНОЙ ФУНКЦИИ **23**

ПРИЛОЖЕНИЕ 2. ПРОГРАММНЫЙ КОД ПОТОКОВОЙ ФУНКЦИИ, РЕАЛИЗУЮЩЕЙ РАЗДЕЛЕНИЕ ЧИСЕЛ НА ОДИНАКОВЫЕ ЧАСТИ ПО КОЛИЧЕСТВУ ПОТОКОВ **24**

ПРИЛОЖЕНИЕ 3. ПРОГРАММНЫЙ КОД ПОТОКОВОЙ ФУНКЦИИ, РЕАЛИЗУЮЩЕЙ РАЗДЕЛЕНИЕ ЧИСЕЛ МЕЖДУ ДВУМЯ ПОТОКАМИ НА ЧЕТНЫЕ И НЕЧЕТНЫЕ 24

ПРИЛОЖЕНИЕ 4. ПРОГРАММНЫЙ КОД ПОТОКОВОЙ ФУНКЦИИ, РЕАЛИЗУЮЩЕЙ РАЗДЕЛЕНИЕ МНОЖЕСТВА ЧИСЕЛ НА НЕБОЛЬШИЕ ГРУППЫ..... 25

Введение

Одним из способов достижения параллелизма выполнения программы является создание потоков. Это обычная практика при решении вычислительно трудоемких задач. Несмотря на внешнюю простоту многопоточного программирования, на разных этапах разработки многопоточных приложений возникает множество вопросов, связанных с реализацией, проверкой правильности работы программы, масштабируемостью при переходе на систему с общей памятью, имеющую большее количество доступных ядер и другие. Различные инструменты разработки параллельных программ, в частности, Intel Parallel Studio, помогают решить такого рода вопросы и значительно повысить эффективность параллельного приложения.

Благодарности

Авторы выражают благодарность за идею и предоставленные фрагменты текста Кириллу Корнякову и Александру Шишкову.

1. Методические указания

1.1. Цели и задачи работы

Цель данной лабораторной работы – рассмотрение на примере задачи разложения чисел на простые множители некоторых вопросов, возникающих при распараллеливании алгоритмов на системах с общей памятью, и приобретение навыков анализа и сравнения различных подходов к распараллеливанию с помощью инструментов Intel Parallel Studio.

Данная цель предполагает решение следующих основных задач:

1. Изучение простейшего алгоритма разложения чисел на простые множители[1].
2. Многопоточная реализация данного алгоритма, которая предусматривает разделение множества факторизуемых чисел на равные части по количеству потоков. Поиск ошибок работы с памятью и ошибок многопоточности с помощью инструмента Intel Parallel Inspector.
3. Анализ эффективности и степени параллелизма полученной реализации средствами Intel Parallel Amplifier.
4. Поиск других подходов к распределению нагрузки между потоками для повышения эффективности приложения.

1.2. Структура работы

В начале работы приводится краткое описание возможностей инструмента Intel Parallel Studio. Далее рассматривается задача разложения множества чисел на простые множители. В процессе выполнения лабораторной работы реализуется несколько многопоточных версий алгоритма в соответствии с различными подходами к распределению нагрузки между потоками. Демонстрируются типичные ошибки работы с памятью и ошибки многопоточности, которые детектируются инструментами пакета **Intel Parallel Studio**. Также с помощью **Intel Parallel Studio** оценивается эффективность и степень параллелизма каждой из реализаций.

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Таблица 1. Тестовая инфраструктура

Процессор	2 четырехъядерных процессора Intel Xeon E5520 (2.27 GHz)
Память	16 Gb
Операционная система	Microsoft Windows 7
Среда разработки	Microsoft Visual Studio 2008
Компилятор, профилировщик, отладчик	Intel Parallel Studio SP1

[1] Рассматриваемый алгоритм является малоэффективным, его использование в лабораторной работе носит иллюстративный характер. При решении практических задач, как правило, применяются другие алгоритмы, например, алгоритмы Полларда или Диксона ([1-3]).

2. Инструмент Intel Parallel Studio

Intel Parallel Studio (далее **IPS**) – это инструмент разработки и анализа эффективности параллельных приложений. В состав **IPS** входит несколько продуктов, каждый из которых может быть проинтегрирован в среду разработки **Microsoft Visual Studio**:

1. **Intel Parallel Advisor** предоставляет набор решений для параллельной реализации алгоритмов.
2. **Intel Parallel Composer** предназначен для быстрого подключения средств параллельной разработки на основе языка C/C++ с использованием набора библиотек с функциями многопоточного программирования. Данный пакет включает:
   - оптимизирующий компилятор **Intel C++ Compiler** с поддержкой OpenMP 3.0;
   - библиотеку вычислительных примитивов **Intel IPP**;
   - библиотеку разработки параллельных программ **Intel TBB**.
3. **Intel Parallel Inspector** обеспечивает поиск ошибок в коде приложений для любых моделей параллельного программирования. Инструмент позволяет находить два класса ошибок:
   - ошибки многопоточности (threading errors): конкурирующий доступ к данным и взаимные блокировки;
   - ошибки работы с памятью (memory errors): утечки памяти, доступ к несуществующим адресам, освобождение памяти по несуществующему адресу, чтение неинициализированной памяти и др.
4. **Intel Parallel Amplifier** – это средство анализа производительности приложения, которое включает:
   - анализ «горячих точек» (hotspot) – позволяет определить места в программе (функции и операции), в которых тратится больше всего вычислительных ресурсов, а также отследить стек вызовов до «горячих» функций;
   - анализ степени параллелизма (concurrency) – оценка эффективности параллельного кода с точки зрения использования ресурсов процессора. Анализ степени параллелизма позволяет оценить прирост производительности при увеличении числа исполнительных устройств, например, при переходе от 4-ядерной системы к 8-ядерной;

- анализ блокировок и ожиданий (locks and waits) – выявляет проблемы блокирования потоков, замедляющие скорость исполнения программы, и идентифицирует соответствующие объекты синхронизации.

3. Задача разложения чисел на простые множители

Лабораторная работа построена на задаче факторизации (разложения на простые множители) чисел из диапазона от 1 до *N*. Используемый алгоритм базируется на попытке деления факторизуемого числа на каждое из меньших его чисел [2]. Если остаток от деления на некоторый множитель равен нулю, то этот множитель запоминается, частное становится делимым, после чего производится повторная попытка деления на это же число. Алгоритм завершает свою работу, когда частное от очередного деления равно единице.

Рассмотрим работу алгоритма для одного числа. Допустим, что необходимо определить простые множители числа 12. Для этого перебираются числа, меньшие 12, начиная с 2, и выполняется последовательность операций деления:

12 / 2 = 6 // остаток равен 0, пробуем делить еще раз на 2

6 / 2 = 3 // остаток равен 0, пробуем делить еще раз на 2

3 / 2 = 1,5 // остаток отличен от 0, рассматриваем следующее число,

// меньшее 12

3 / 3 = 1 // частное равно 1, алгоритм останавливает работу

Псевдокод последовательного алгоритма выглядит следующим образом:

```
1. for i = 1 to N
2.   number ← i;
3.   for j = 2 to i
4.     if (number == 1) break;
5.     r ← number % j;
6.     if (r == 0)
7.       number ← number / j;
8.       save_divisor(i, j);
9.       j ← j - 1;
```

В соответствии с приведенным псевдокодом может быть реализована простейшая последовательная версия алгоритма.

4. Разделение множества чисел на одинаковые части по количеству потоков

4.1. Задание 1. Открытие проекта 01_EqualPartition

Откройте проект **01_EqualPartition**, последовательно выполняя следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2008**.
- В меню **File** выполните команду **Open→Project/Solution…**.
- В диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\01_PrimeNumbers**. Дважды кликните по файлу **01_PrimeNumbers.sln** или, выбрав файл, выполните команду **Open** (рис. 1).

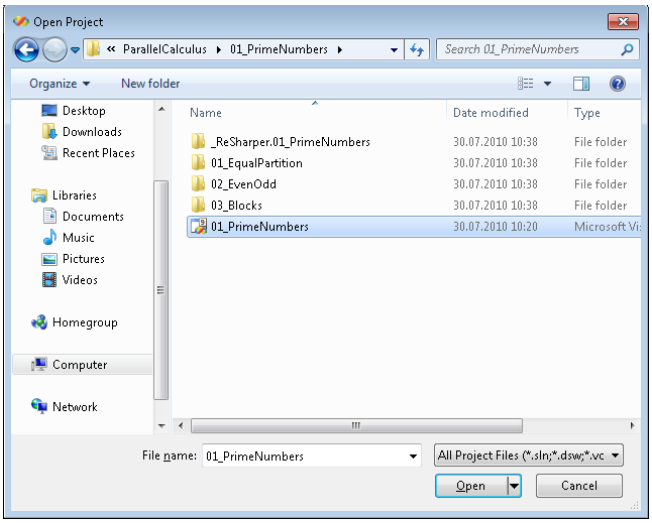

Рис. 1. Диалоговое окно открытия проекта Microsoft Visual Studio

- Сделайте проект **01_EqualPartition** рабочим. Для этого кликните правой кнопкой мыши по проекту и выберите пункт **Set as Startup Project**.
- После открытия решения в окне **Solution Explorer** (Ctrl+Alt+L) щелкните на проекте **01_EqualPartition**, в котором содержится файл исходного кода **EqualPartition.cpp**, как это показано на рис. 2.

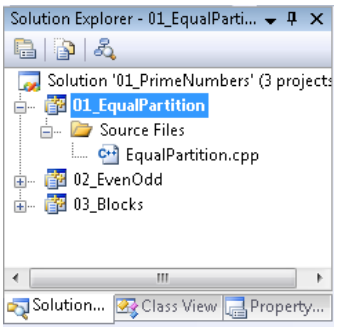

Рис. 2. Окно Solution Explorer

После этих действий в окне рабочей области **Visual Studio** будет открыт программный код. Данный файл содержит подключение необходимых библиотек, объявление констант и переменных, основную функцию **`main()`** и объявление потоковой функции.

Предопределенные константы.

```
// количество факторизуемых чисел
#define NUM_NUMBERS 100000
// количество создаваемых потоков
#define NUM_THREADS 8
```

В программе объявлены две глобальные переменные: массив векторов, который в результате работы будет заполнен простыми множителями, и массив дескрипторов потоков.

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

Далее объявляется потоковая функция. Каждый из рассматриваемых подходов к распараллеливанию предполагает собственную реализацию потоковой функции. На начальном этапе потоковая функция имеет пустую реализацию.

```
DWORD WINAPI factorization(LPVOID)
{
  return 0;
}
```

Основная функция **`main()`** имеет следующую структуру:

- создание потоков, обрабатывающих некоторый набор чисел из диапазона от 1 до *N*;

```
// создание потоков
for (int i = 0; i < NUM_THREADS; i++)
{
  tHandle[i] = CreateThread(NULL, 0, factorization,
                 &i, 0, NULL);
}
```

- ожидание завершения исполнения потоков;

```
WaitForMultipleObjects(NUM_THREADS, tHandle, TRUE,
  INFINITE);
```

- вывод простых множителей некоторых чисел из рассматриваемого диапазона и освобождение динамически выделенной памяти.

На данном этапе можно скомпилировать проект, выполнив команду **Project Only→Rebuild Only 01_EqualPartition** в меню **Build**. Если приложение скомпилировано успешно (в нижней части окна **Visual Studio** появилось сообщение "Rebuild All: 1 succeeded, 0 failed, 0 skipped"), нажмите клавишу F5 или выполните команду **Start Debugging** пункта меню **Debug**, чтобы осуществить запуск приложения. Либо нажмите сочетание клавиш Ctrl+F5, чтобы запустить приложение не в режиме отладчика. В результате программа выведет несколько случайным образом выбранных чисел из диапазона от 1 до *N*. Для каждого числа в коде предусмотрен вывод на экран его разложения на простые множители. Пока этот вывод ничего не делает, поскольку простые множители не найдены. Переходим к реализации потоковой функции.

4.2. Задание 2. Реализация параллельного алгоритма. Поиск ошибок работы с памятью и ошибок многопоточности

Простейший подход к распараллеливанию в задаче разложения чисел из диапазона от 1 до *N* состоит в том, чтобы разделить множество факторизуемых чисел на равные части по числу потоков. На рис. 3 показан пример распределения чисел при создании четырех потоков.

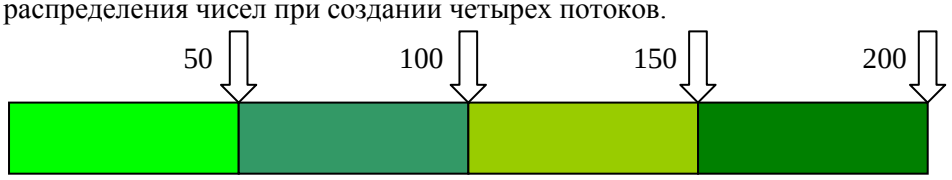

Рис. 3. Распределение нагрузки между потоками – вариант 1

В этом случае в потоковой функции необходимо определить по идентификатору потока начало и конец блока чисел, которые факторизуются данным потоком. Идентификатор потока наиболее разумно передавать в качестве параметра потоковой функции (что мы и попытались сделать выше, передав в потоковую функцию адрес счетчика цикла **`i`**). Затем для каждого числа необходимо построить набор простых множителей в соответствии с алгоритмом, описанным в предыдущем разделе. Ниже представлены формулы вычисления начала и конца обрабатываемого блока.

```
// tid – идентификатор потока
1. start ← (NUM_NUMBERS / NUM_THREADS) * tid + 1;
```

```
2. end   ← (NUM_NUMBERS / NUM_THREADS) * (tid + 1) + 1;
```

Реализуйте описанный подход. Скомпилируйте проект и убедитесь, что при запуске программы происходит падение.

Проверим программу на наличие ошибок работы с памятью и ошибок многопоточности, используя инструмент **Intel Parallel Inspector**.

Чтобы использовать **Intel Parallel Inspector** соберите **Debug**-версию программы. Для поиска ошибок многопоточности на панели инструментов выберите режим **Threading errors** (рис. 4).

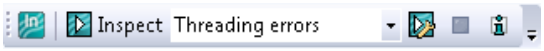

Рис. 4. Панель инструментов Intel Parallel Inspector (режим Threading errors)

После нажатия на кнопку **Inspect**, задайте режим (глубину) анализа **Analysis Level: High** (рис. 5).

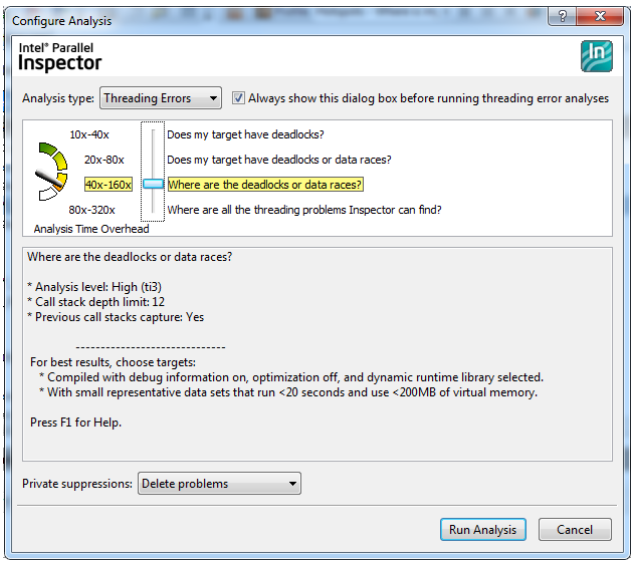

Рис. 5. Окно выбора режима анализа

Запустите инспектор посредством нажатия на кнопку **Run Analysis**. Отметим, что процесс анализа существенно увеличивает время исполнения программы. Если ожидание результатов затягивается, рекомендуется остановить процесс и уменьшить объем обрабатываемых данных.

Intel Parallel Inspector обнаружит в нашем коде одну из типичных «параллельных» ошибок – гонку данных (рис. 6), которая и является причиной падения программы.

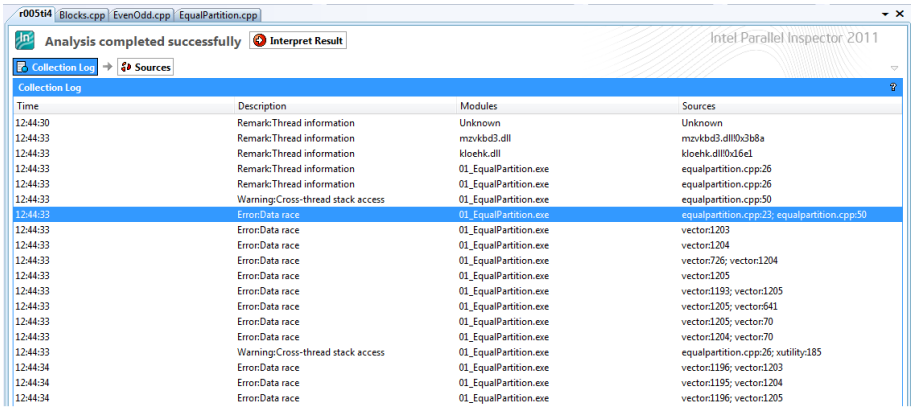

Рис. 6. Intel Parallel Inspector – гонка данных

При обнаружении ошибки инструмент явно указывает номер строки кода, в которой она имеет место и позволяет перейти к режиму просмотра кода по двойному щелчку (рис. 7).

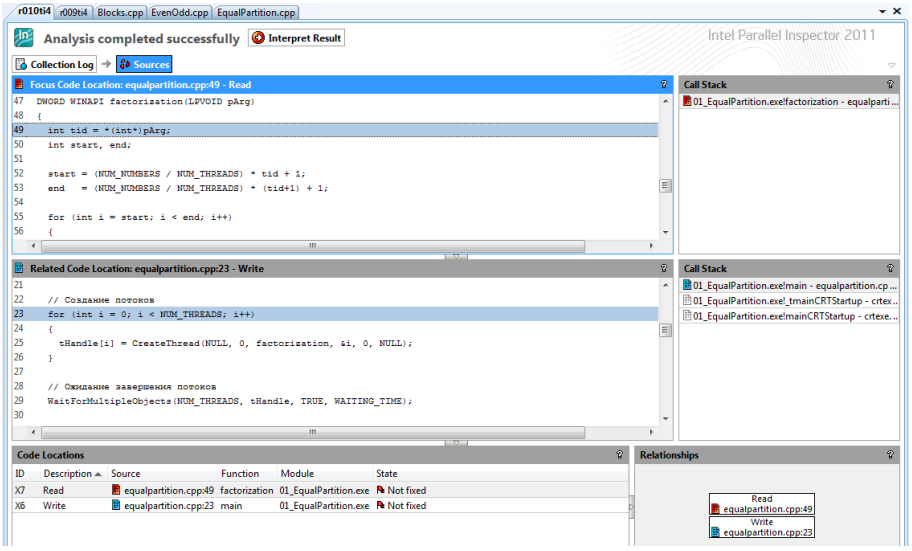

Рис. 7. Неверная передача номера потока

Глядя на строку с ошибкой, нетрудно понять, в чем она состоит. Счетчик цикла, который мы использовали для передачи номера потока, содержит правильное значение только до тех пор, пока не начнется следующая итерация (то есть до окончания работы функции **`CreateThread()`**). После чего его значение меняется, и потоки при использовании этой переменной получают неверные границы цикла в потоковой функции. Более того, последние потоки будут работать со значением **`i`** равным 8, а значит, будут

обращаться за границы созданного нами STL-вектора **divisors**. Собственно, это и приводит к падению программы.

Исправить ошибку можно, создав в функции **main()** массив для хранения номеров потоков и передавая в потоковую функцию адрес его элементов. В представленном ниже коде необходимые добавления выделены полужирным начертанием.

```
int *tNum = new int[NUM_THREADS];
// создание потоков
for (int i = 0; i < NUM_THREADS; i++)
{
  tNum[i] = i;
  tHandle[i] = CreateThread(NULL, 0, factorization,
                  &tNum[i], 0, NULL);
}
```

Соберите проект и убедитесь, что падение при запуске программы исчезло, а результаты работы стали корректными.

Теперь проверим ситуацию с памятью. На панели инструментов **Intel Parallel Inspector** выберите режим **Memory errors** (рис. 8) для поиска ошибок работы с памятью. Далее задайте режим анализа **Analysis Level: Extreme**.

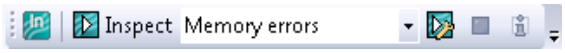

Рис. 8. Панель инструментов Intel Parallel Inspector (режим Memory errors)

Наиболее распространенная ошибка при работе с памятью – это утечка памяти (рис. 9). В представленном коде в функции **main()** допущена типовая ошибка данного вида – не освобождена память, выделенная под массив номеров потоков **tNum**.

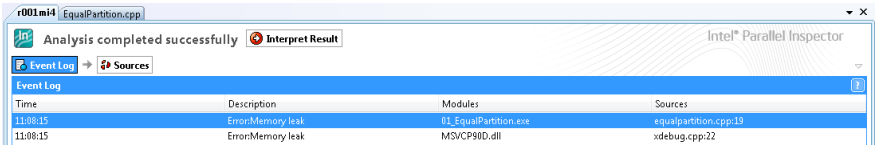

Рис. 9. Intel Parallel Inspector – утечка памяти

При обнаружении такой ошибки инструмент явно указывает номер строки кода, в которой динамически выделяется память, впоследствии не освобождаемая (рис. 10).

```
17  void main()
18  {
19    int *tNum = new int[NUM_THREADS];
20
```

Рис. 10. Выделение блока памяти

Исправьте ошибку, добавив в конце функции **`main()`** освобождение памяти. Еще один возможный вариант исправления – изменить объявление массива **`tNum`** на статическое.

```
delete [] tNum;
```

4.3. Задание 3. Анализ эффективности («горячие» точки)

Следующий шаг после получения корректной параллельной версии – анализ эффективности. На данном этапе необходимо определить функции, в которых программа проводит большую часть времени исполнения. Именно оптимизация этих функций даст наибольший эффект в повышении производительности программы. Воспользуемся следующим инструментом из пакета **IPS** – **Intel Parallel Amplifier**.

1. Соберите **Release**-версию проекта. Выберите **Hotspots**-режим работы **Intel Parallel Amplifier** (рис. 11). Для запуска процесса профилировки необходимо нажать на кнопку **Profile**.

Рис. 11. Панель инструментов Intel Parallel Amplifier

2. Определите функцию, которая работает дольше остальных (рис. 12). В данном случае очевидно, что самой «медленной» будет потоковая функция, поскольку в ней сосредоточена вся вычислительная нагрузка. Дважды кликнув на этой функции, можно увидеть, что наиболее трудоемкой операцией является операция сравнения остатка от деления факторизуемого числа (рис. 13). Как следствие, необходимо минимизировать количество таких операций.

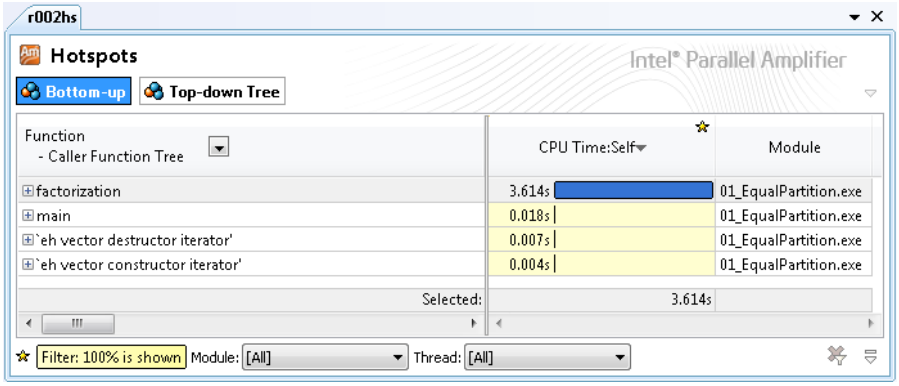

Рис. 12. Результаты работы Intel Parallel Amplifier

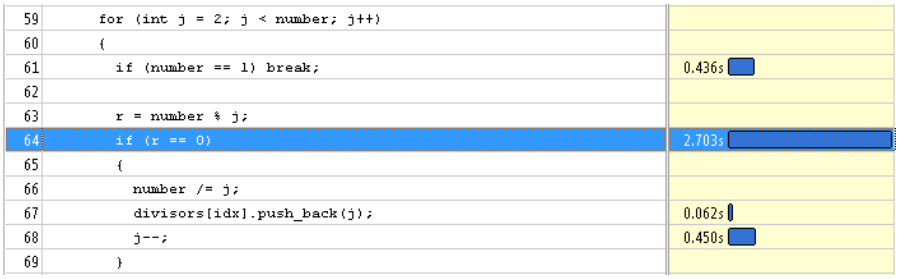

Рис. 13. Наиболее трудоемкая операция

3. Выполним алгоритмическую оптимизацию. Реализованный алгоритм является избыточным с точки зрения количества операций, т.к. простые множители числа находятся в диапазоне от двух до половины значения факторизуемого числа. Внесите соответствующие изменения в код, запустите **Intel Parallel Amplifier** в режиме **Hotspots** и определите текущее время работы потоковой функции (рис. 14).

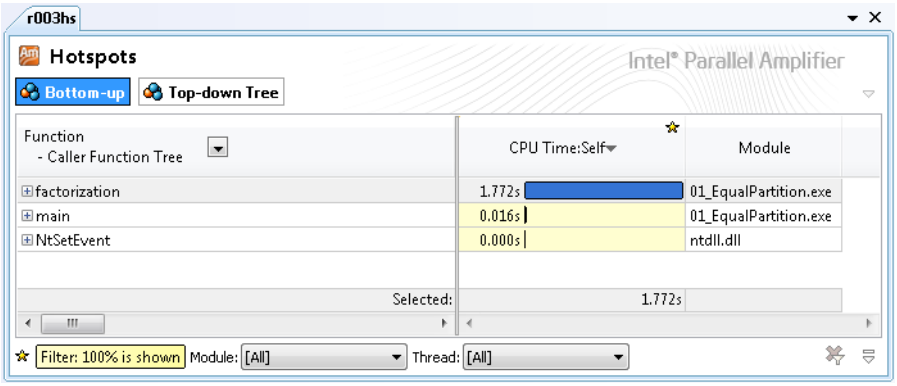

Рис. 14. Intel Parallel Amplifier – результаты после алгоритмической оптимизации

В результате применения алгоритмической оптимизации суммарное время работы приложения уменьшилось приблизительно на 49% (рис. 15).

Понятно, что алгоритмическая оптимизация на этом не заканчивается. При практическом решении данной задачи нужно использовать другие алгоритмы, о чем мы уже упоминали. Здесь же мы остановимся на данном шаге и перейдем к исследованию степени параллелизма разработанной версии.

Рис. 15. Сравнение суммарного времени факторизации

4.4. Задание 4. Оценка степени параллелизма

Для оценки эффективности программы с точки зрения степени ее параллелизма запустим **Intel Parallel Amplifier** в режиме **Concurrency**. На рис. 16 показан результат запуска текущей реализации в восемь потоков.

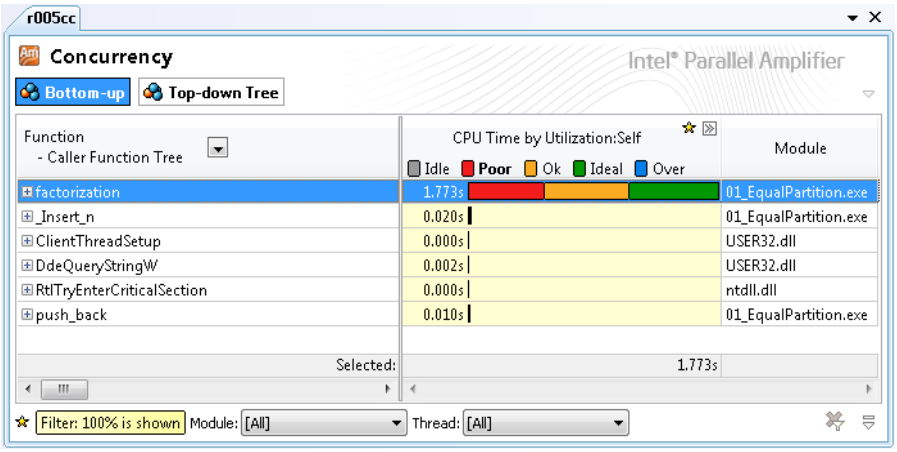

Рис. 16. Intel Parallel Amplifier – режим Concurrency

Для каждой функции построена временная шкала, которая отражает полноту использования процессорных ресурсов при выполнении программы. Из рисунка можно сделать вывод, что практически треть времени работы программа использует не все предоставляемые ресурсы (об этом свидетельствует наличие красного цвета на шкале). Данный факт говорит о том, что в процессе реализации была использована неэффективная схема распараллеливания.

Обратите внимание на диаграмму, представленную в окне **Summary** результатов анализа в режиме **Concurrency** (рис. 17).

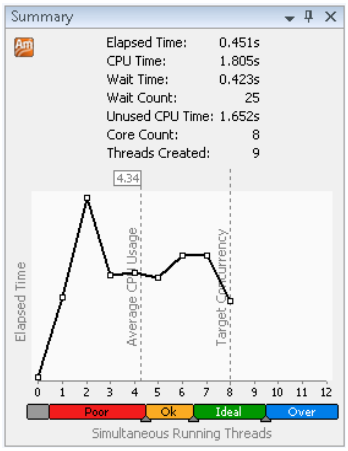

Рис. 17. Диаграмма распределения времени работы программы по числу активных потоков

На диаграмме показано, сколько времени в процессе исполнения программы активно количество потоков, отложенное по оси абсцисс. В идеальном случае программа все время работы должна проводить в 8 потоков, тогда график будет близок к оси абсцисс от 0 до 7 и будет иметь единственный пик на отметке 8 потоков. Однако на практике полностью исключить последовательные участки кода невозможно, поэтому при наилучшей реализации график должен иметь монотонно возрастающий характер. В данном случае мы видим, что текущая реализация малоэффективна. Больше всего времени программа работает в два потока, а время, когда задействованы все 8 потоков является наименьшим из всех. Это означает, что вычислительная нагрузка между потоками распределена неравномерно. Необходимо искать другие походы к распределению множества чисел между потоками.

5. Разделение множества чисел с чередованием

5.1. Задание 1. Открытие проекта 02_EvenOdd

Откройте проект **02_EvenOdd**, последовательно выполняя следующие шаги:

- Сделайте проект **02_EvenOdd** рабочим. Для этого кликните правой кнопкой мыши по проекту и выберите пункт **Set as Startup Project**.
- В окне **Solution Explorer** (Ctrl+Alt+L) дважды щелкните на проекте **02_EvenOdd** и откройте файл исходного кода **EvenOdd.cpp**.

Скомпилируйте проект, выполнив команду **Project Only→Rebuild Only 02_EvenOdd** в меню **Build**. Нажмите клавишу F5 или выполните команду **Start Debugging** пункта меню **Debug**, чтобы выполнить запуск приложения. Как и ранее в результате программа выведет несколько случайным образом выбранных чисел из диапазона от 1 до *N*. Чтобы получить для этих чисел простые множители, необходимо реализовать потоковую функцию в соответствии с очередным подходом.

5.2. Задание 2. Реализация параллельного алгоритма

Распределение чисел непрерывными блоками по числу потоков ведет к тому, что чем больше номер потока, тем большие числа ему достаются для проверки и тем больше времени это требует. Попытаемся улучшить балансировку, разделив множество факторизуемых чисел с чередованием – число обрабатывается тем потоком, идентификатор которого совпадает с остатком от деления этого числа на общее количество потоков. На рис. 18 показан пример распределения чисел при создании четырех потоков.

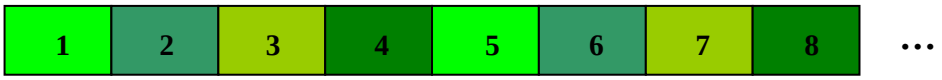

Рис. 18. Распределение нагрузки между потоками – вариант 2

При таком подходе каждый поток, как и ранее, получает набор чисел размером, равным отношению количества чисел в рассматриваемом диапазоне к общему числу потоков. Но теперь значения факторизуемых чисел определяются через идентификатор потока в соответствии с формулой, приведенной ниже. Поиск простых множителей осуществляется аналогично предыдущему случаю.

```
// tid – идентификатор потока
// i – индекс числа в блоке чисел, факторизуемых потоком
1. number = i * NUM_THREADS + tid + 1;
```

Реализуйте описанный подход. Скомпилируйте проект и убедитесь в правильности реализации. Проверьте отсутствие ошибок работы с памятью и ошибок многопоточности, используя **Intel Parallel Inspector**.

5.3. Задание 3. Оценка степени параллелизма

Для оценки степени параллелизма запустите **Intel Parallel Amplifier** в режиме **Concurrency.** На рис. 19 показан результат работы в восемь потоков. Для каждой функции построена временная шкала, которая отражает полноту использования процессорных ресурсов при выполнении программы. Аналогично предыдущей реализации значительную часть времени про-

грамма использует не все предоставляемые ресурсы, о чем свидетельствует наличие красного цвета на шкале.

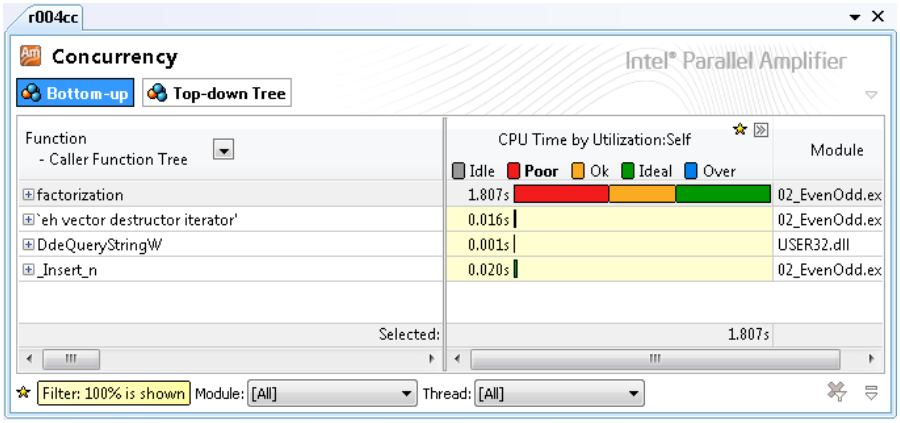

Рис. 19. Intel Parallel Amplifier – режим Concurrency

Если посмотреть на диаграмму **Summary** (рис. 20), можно увидеть, что ситуация чуть улучшилась по сравнению с предыдущей версией. Теперь больше всего времени программа работает в 4 потока, следующий пик приходится на 6 потоков, а время, когда задействованы все 8 потоков, по-прежнему достаточно мало.

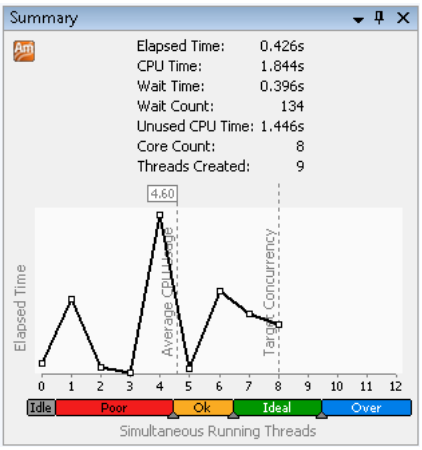

Рис. 20. Диаграмма распределения времени работы программы по числу активных потоков

Таким образом, и в данной версии вычислительная нагрузка распределена между потоками неравномерно. Теперь причина в том, что четные числа факторизуются быстрее нечетных (у них первый же делитель равный двум уменьшает эти числа вдвое), поэтому четыре потока, обрабатывающие четные числа, завершаются раньше. Чтобы обеспечить полноту использо-

вания ресурсов, далее рассматривается еще один подход к распределению чисел между потоками.

6. Разделение множества чисел на группы

6.1. Задание 1. Открытие проекта 03_Blocks

Откройте проект **03_Blocks**, последовательно выполняя следующие шаги:

- Сделайте проект **03_ Blocks** рабочим. Для этого кликните правой кнопкой мыши по проекту и выберите пункт **Set as Startup Project**.
- В окне **Solution Explorer** (Ctrl+Alt+L) дважды щелкните на проекте **03_ Blocks**, в котором содержится файл исходного кода **Blocks.cpp**.

Скомпилируйте проект, выполнив команду **Project Only→Rebuild Only 03_ Blocks** в меню **Build**. Нажмите клавишу F5 или выполните команду **Start Debugging** пункта меню **Debug**, чтобы выполнить запуск приложения. Как и ранее в результате программа выведет несколько случайным образом выбранных чисел из диапазона от 1 до *N*. Чтобы получить для этих чисел простые множители, реализуем потоковую функцию в соответствии с подходом, описанным в следующем разделе.

6.2. Задание 2. Реализация параллельного алгоритма

Последний рассматриваемый в данной работе подход к распределению нагрузки в задаче разложения чисел из диапазона от 1 до *N* состоит в том, чтобы разделить множество факторизуемых чисел на группы и делить группы между потоками чередованием. На рис. 21 показан пример распределения чисел при создании четырех потоков.

Рис. 21. Распределение нагрузки между потоками

При таком подходе каждый поток получает совокупность числовых блоков из разных частей диапазона. Количество блоков равно отношению количества чисел в диапазоне к общему числу потоков, деленному на размер блока (предполагается, что все операции деления дают в остатке ноль). Размер, начало, конец блока вычисляются в соответствии с формулами, приведенными ниже. Оптимальный размер блока (GRAIN_SIZE) определяется экспериментально. Поиск простых множителей для каждого числа в группе осуществляется аналогично последовательному алгоритму.

```
// tid – идентификатор потока
1. numberOfGrains = NUM_NUMBERS / NUM_THREADS / GRAIN_SIZE;
2. for i = 0 to numberOfGrains
3.   begin = (NUM_THREADS * i + tid) * GRAIN_SIZE + 1;
4.   end   = (NUM_THREADS * i + tid + 1) * GRAIN_SIZE + 1;
5. ...
```

Реализуйте описанный подход. Скомпилируйте проект и убедитесь в правильности реализации. Проверьте отсутствие ошибок работы с памятью и ошибок многопоточности, используя **Intel Parallel Inspector**.

6.3. Задание 3. Оценка степени параллелизма

Для оценки степени параллелизма текущей реализации запустите **Intel Parallel Amplifier** в режиме **Concurrency**. На рис. 22 показан результат запуска на восьми потоках при размере блока равном 900. В отличие от предыдущих реализаций большую часть времени программа использует все предоставляемые ресурсы, о чем свидетельствует зеленый цвет, который заполняет практически всю шкалу, соответствующую потоковой функции.

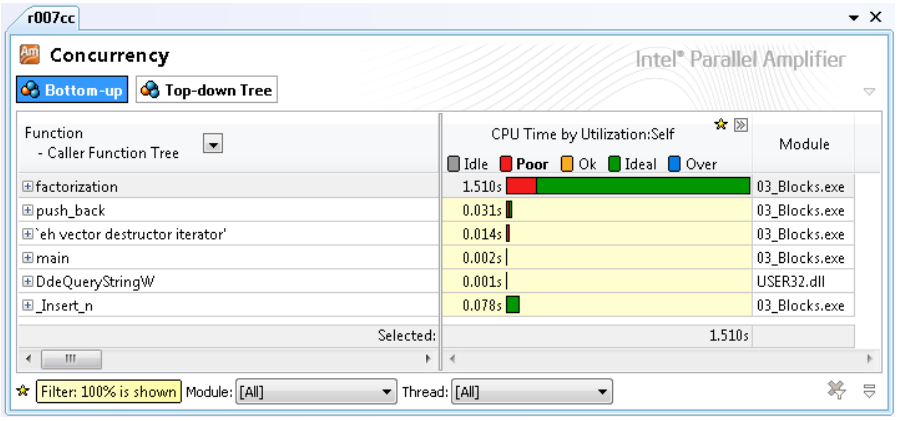

Рис. 22. Intel Parallel Amplifier – режим Concurrency

Если обратиться к графику в окне **Summary** (рис. 23), можно увидеть, что большую часть времени программа работает в 8 потоков. Пик, приходящийся на один активный поток, объясняется накладными расходами на создание потоков и их синхронизацию, – фактически это код функции **`main()`**. С увеличением количества обрабатываемых чисел этот пик будет уменьшаться. Характер графика несколько изменится, если закомментировать в функции **`main()`** код, выполняющий вывод разложения случайно выбранных чисел, однако общая тенденция сохранится.

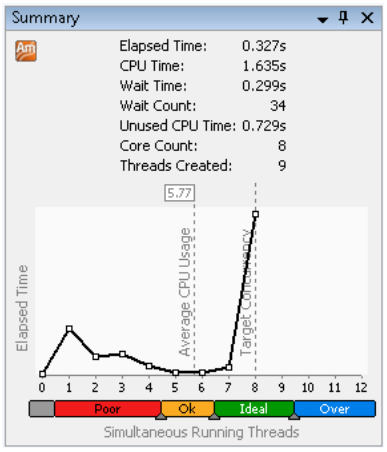

Рис. 23. Диаграмма распределения времени работы программы по числу активных потоков

Таким образом, данный подход является наиболее эффективным, но требует нахождения оптимального размера обрабатываемого блока.

7. Контрольные вопросы

1. Какие типы ошибок позволяет определить инструмент **Intel Parallel Inspector**?
2. Какие виды оптимизации можно применить при обнаружении «горячих точек»?
3. Наличие какого цвета при оценке степени параллелизма в инструменте **Intel Parallel Amplifier** свидетельствует о проблемах в распараллеливании приложения?
4. Опишите качественную структуру графика распределения времени работы программы по числу активных потоков для программы, которая эффективно использует предоставляемые ресурсы.

8. Дополнительные задания

В данном разделе приводятся дополнительные задания для получения более полного представления о проблемах, которые возникают при распараллеливании алгоритмов на системах с общей памятью, закрепления навыков анализа результатов, предоставляемых пакетом **Intel Parallel Studio**.

1. Подберите оптимальный размер блока при реализации последнего подхода.
2. Реализуйте алгоритм Полларда или Диксона для определения простых множителей числа ([1-3]).
3. Проанализируйте эффективность описанных подходов к распределению нагрузки между потоками применительно к задаче факторизации набора чисел из диапазона от 1 до *N* при условии, что для получения простых множителей числа используется алгоритм, реализованный в процессе выполнения предыдущего самостоятельного задания. Используйте инструменты пакета **Intel Parallel Studio**.
4. Сравните эффективность и степень параллелизма реализаций, полученных при выполнении основного и дополнительных заданий лабораторной работы.

9. Литература

1. Василенко О.Н. Теоретико-числовые алгоритмы в криптографии. 2003. – М.: МЦНМО, 2003.Стр. 78-83.
2. Кнут Д. Искусство программирования. – М.: Вильямс, 2007. Т.2, с. 465-468.
3. Черемушкин А.В. Лекции по арифметическим алгоритмам в криптографии. – М.: МЦНМО, 2002. Стр. 77-80.

Приложение 1. Программный код основной функции

```
#define NUM_NUMBERS 100000
#define NUM_THREADS 8
vector<int> divisors[NUM_NUMBERS+1];
HANDLE tHandle[NUM_THREADS];
// Объявление потоковой функции
DWORD WINAPI factorization(LPVOID);
void main()  {
  int *tNum = new int[NUM_THREADS];
  // Создание потоков
  for (int i = 0; i < NUM_THREADS; i++)  {
    tNum[i] = i;
    tHandle[i] = CreateThread(NULL, 0, factorization,
                              &tNum[i], 0, NULL);
  }
  // Ожидание завершения потоков
  WaitForMultipleObjects(NUM_THREADS, tHandle, TRUE,
                         INFINITE);
```

```
  // Вывод простых множителей произвольных 10 чисел
  for (int i = 0; i < 10; i++)  {
    int randomIdx = 1 + rand() % NUM_NUMBERS;
    cout << randomIdx << ":\t";
    int size=static_cast<int>(divisors[randomIdx].size());
    for (int j = 0; j < size; j++)    {
      cout << divisors[randomIdx][j] << "\t";
    }
    cout << endl;
  }
  delete []tNum;
}
```

Приложение 2. Программный код потоковой функции, реализующей разделение чисел на одинаковые части по количеству потоков

```
DWORD WINAPI factorization(LPVOID pArg) {
  int tid = *(int*)pArg;
  int start, end, r;
  start = (NUM_NUMBERS / NUM_THREADS) * tid + 1;
  end   = (NUM_NUMBERS / NUM_THREADS) * (tid+1) + 1;
  for (int i = start; i < end; i++)  {
    int number = i;
    int idx = number;
    for (int j = 2; j < idx; j++)  {
      if (number == 1) break;
      r = number % j;
      if (r == 0)  {
        number /= j;
        divisors[idx].push_back(j);
        j--;
      }
    }
  }
  return 0;
}
```

Приложение 3. Программный код потоковой функции, реализующей разделение чисел между двумя потоками на четные и нечетные

```
DWORD WINAPI factorization(LPVOID pArg)  {
  int tid = *(int*)pArg;
```

```
  int r;
  for (int i = 0; i < NUM_NUMBERS / NUM_THREADS; i++)  {
    int number = i * NUM_THREADS + tid + 1;
    int idx = number;
    for (int j = 2; j < idx / 2; j++)  {
      if (number == 1) break;
      r = number % j;
      if (r == 0)  {
        divisors[idx].push_back(j);
        number /= j;
        j--;
      }
    }
  }
  return 0;
}
```

Приложение 4. Программный код потоковой функции, реализующей разделение множества чисел на небольшие группы

```
DWORD WINAPI factorization(LPVOID pArg)  {
  int tid = *(int*)pArg;
  int numberOfGrains = NUM_NUMBERS/NUM_THREADS/GRAIN_SIZE;
  int begin, end;
  int number, r;
  for (int g = 0; g < numberOfGrains; g++)  {
    begin = (NUM_THREADS*g + tid)*GRAIN_SIZE + 1;
    end   = (NUM_THREADS*g + tid + 1)*GRAIN_SIZE + 1;
    for (int i = begin; i < end; i++)  {
      number = i;
      int idx = number;
      for (int j = 2; j < idx / 2; j++)  {
        if (number == 1) break;
        r = number % j;
        if (r == 0)  {
          divisors[idx].push_back(j);
          number /= j;
          j--;
        }
      }
    }
  }
  return 0;
}
```