

**Практические рекомендации по
распараллеливанию с помощью OpenMP и
измерению ускорения.**

**Ошибки при многопоточном
программировании.**

Распределение заданий между потоками

ЛЕКЦИЯ №4

OpenMP

1. **Практические рекомендации по распараллеливанию с помощью OpenMP и измерению ускорения.**
2. **Ошибки при многопоточном программировании.**
3. **Презентация материалов по OpenMP**
4. **Распределение заданий между потоками**

1. Практические рекомендации по распараллеливанию с помощью OpenMP и измерению ускорения.

- 1.1. Время вычислений в параллельном регионе должно быть больше, чем время, затраченное на создание параллельного региона

- 1.2. При входе в первый параллельный регион «накладные расходы» намного больше, чем при входах в следующие параллельные регионы

Потоки в современных Windows

- *Процесс* представляет выполняющийся экземпляр программы. Он имеет собственное адресное пространство, где содержаться его код и данные.
- *Процесс* должен содержать минимум как один *поток*, так как именно он, а не процесс, является единицей планирования (данная операционная система относится к *системам разделения времени*, т.е. каждой единице предоставляется *квант* процессорного времени).

4

- *Процесс* может создавать несколько потоков выполняемых в его адресном

Классы потоков. Потоки OpenMP

Основные состояния потока. «Накладные расходы»

Неизбежные «накладные расходы» в многопоточной программе с несколькими параллельными регионами

- Создание потоков – самые большие «накладные расходы»
- На «остановлен» - «на процессоре» - «накладные расходы» намного меньше, чем на создание (вход – выход в параллельную секцию)
- На «ожидание» - «на процессоре» «накладные расходы» намного меньше, чем на «остановлен» - «на процессоре» (критическая секция – зато чаще встречается, чем вход в параллельную секцию)

1. 1. Время вычислений в параллельном регионе должно быть больше, чем время, затраченное на создание параллельного региона

Это время может быть определено из работы цикла вида

```
for (i = 1; i < treeData.N; i++)  
    {  
#pragma omp parallel  
    {  
    }  
    }  
8
```


1.2. При входе в первый параллельный регион «накладные расходы» намного больше, чем при входах в следующие параллельные регионы

- Либо полное время параллельных вычислений должно быть больше кванта операционной системы (для систем разделения времени, в том числе Windows)
 - Либо перед тестируемым участком поставить директиву по созданию пустого параллельного региона
- большая часть «накладных расходов» будет «локализована» в ней
- условие выбора величины кванта – чтобы работа процессов «приносила больше пользы», чем «накладные расходы» на их инициирование)

Тестируемый код – проект `time_parallel` – ускорение как функция полного времени работы программы – последовательный код

```
start = rdtsc();  
for (j=1; j <=M; j++)  
  for (i = 1; i < treeData.N; i++)  
    if (treeData.Path[i] > limit)  
      {  
        Weight_PathMin = treeData.max+limit;  
        if (Weight_PathMin > limit)  
          {  
            replace_number = treeData.Weight[i];  
          }  
      }  
stop = rdtsc()-start;
```

1

0

Тестируемый код – проект `time_parallel` – ускорение как функция полного времени работы программы – параллельный код (участок в main)

```
start = rdtsc();
for (j = 1; j <=M; j++)
{ omp_set_num_threads (num_threads);
#pragma omp parallel private(id)
  { id = omp_get_thread_num();
    Path_Min[id] = Search_minimum (id, treeData.max,
treeData.Path, treeData.N, num_threads);
  }
MinimPath = treeData.max;
for (i = 0; i < num_threads; i++)
  if(Path_Min[i] < MinimPath) MinimPath = Path_Min[i];
}
stop = rdtsc()-start;
```

Тестируемый код – проект `time_parallel` – ускорение как функция полного времени работы программы – параллельный код

(функция, выполняемая двумя потоками, к ней - обращение из параллельной секции)

```
int Search_minimum(int myID, int minimum, int*mas,
    const int n, const int num_threads )
{ int i, Initial, Final;
    int ID;  int N_Section = n/num_threads;
    Initial = myID*N_Section;  Final = (myID + 1)*N_Section;

    for (i = Initial; i< Final; i++)
        if (mas[i] <= minimum)  minimum = mas[i];

    return minimum;
1
2
}
```

Требования на выбор предельных значений переменных внешнего и внутреннего цикла

1. Внутренний цикл: treeData.N определяется из условия:
 $t(\text{внутреннего цикла}) \gg t(\text{входа в многопоточный регион})$

2.1. Внешний цикл: при фиксированном treeData.N должно быть такое M , что

$t(\text{полное}) \sim \text{или} > t(\text{кванта})$

2.2. Либо M любое, но перед тестируемым циклом стоит директива по созданию параллельного региона

1

3

Задание 1. Проект time_parallel. Зависимость ускорения от M

1. Для запуска последовательного варианта аргументы в командной строке:

- 1) 5 10 10000 1 3
- 2) 5 10 10000 2 3
- 3) 5 10 10000 10 3
- 4) 5 10 10000 100 3
- 5) 5 10 10000 1000 3
- 6) 5 10 10000 100000 3

1. Для запуска параллельного варианта аргументы в командной строке:

- 1) 5 10 10000 1 5 2
- 2) 5 10 10000 2 5 2
- 3) 5 10 10000 10 5 2
- 4) 5 10 **1** 10000 100 5 2
- 5) 5 10 **4** 10000 1000 5 2
- 6) 5 10 10000 100000 5 2

Задание 2. Проект time_parallel. Зависимость ускорения от M

Демонстрация того, что все «накладные расходы» сосредоточены в первом создаваемом многопоточном регионе (в этом варианте – вход в три одинаковых параллельных региона):

Запуск с аргументами командной строки

```
5 10 10000 1 11 2
```

1
5

2. Ошибки при многопоточном программировании

1. Конфликты «запись - запись» - два потока пишут в одну переменную

(«а ля» два рецензента правят один экземпляр статьи и друг друга одновременно – кто успеет быстрее «гонки данных»)

2. Тупики или «зависания» или «lock» (один поток захватил ресурс и не отдает другим – программа может висеть бесконечно)

- Живой
- Мертвый

3. **1** Избыточное применение параллельных конструкций – меньше ускорение, хотя программа работает правильно
6

Ошибки, которые находит ThreadChecker при программировании в OpenMP

1. Конфликты «запись - запись» - два потока пишут в одну переменную

- Значение этой переменной зависит от того, «кто успел быстрее»
- Кроме того, большие «накладные расходы» - резкое увеличение времени работы

3. Презентация материалов по OpenMP

3.1. Курс Гергеля

3.2. Материалы тренингов Intel

3.1. Курс Гергеля

- Обзор методов многопоточного программирования для простейших алгоритмов
 - умножение вектора на вектор
 - матрицы на матрицу
 - решение систем линейных уравнений
 - сортировки
- Обзор основных конструкций OpenMP
- Особенность курса
 - в основном обзорный характер, «охват материала»
 - акцент – на рассмотрении алгоритмов

3.2. Материалы тренингов Intel

Преобладающая особенность – все показывается на одной задаче

- Параллельный алгоритм
- Параллельные конструкции OpenMP
- Методика создания многопоточного приложения
- Интерфейс и возможности отладчиков

Акцент на процесс создания и технологию параллельного программирования

4. Распределение заданий между потоками

По материалам тренинга Intel, проведенного для преподавателей вузов в апреле 2006 г.

Цели и задачи

- Научиться технике распараллеливания последовательного кода на основе OpenMP
- Применять в цикле разработки инструменты Intel
- Оценивать максимально возможное ускорение многопоточной программы по закону Амдала

Содержание

- Стандартный цикл разработки
- Изучаемый пример: генерация простых чисел
- Как повысить эффективность вычислений

2

3

Определение параллелизма

- Два или более процесса или потока выполняются одновременно

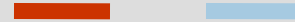
Виды параллелизма для архитектур, поддерживающих потоки

- *Множество процессов*

Взаимодействие между процессами
(Inter-Process Communication (IPC))

Закон Амдала

Оценка «сверху» для ускорения параллельной программы по закону Амдала



Процессы и потоки



Потоки – «плюсы» и «минусы»

«Плюсы»

- Позволяют повысить производительность и полнее использовать системные ресурсы
 - Даже в однопроцессорной системе – для «скрытия» латентности и повышения производительности
- Взаимодействие через разделяемую (общую) память более эффективно

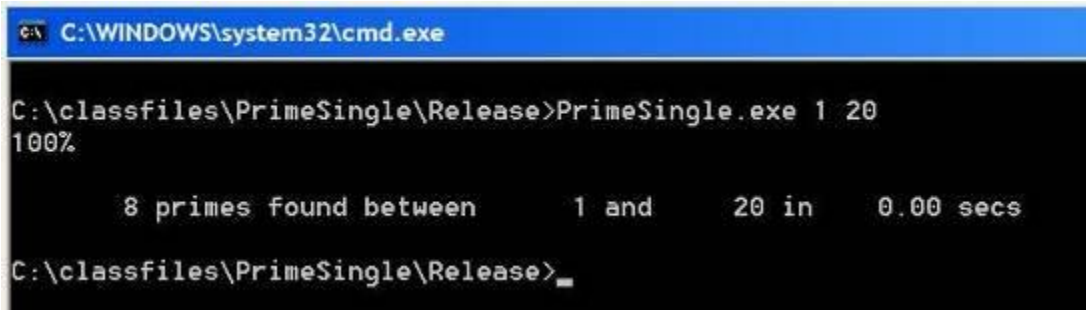
«Минусы»

- Возрастает степень сложности
 - Сложность в отладке приложений
- («гонки данных» (конфликты «запись-запись» и т. д.), тупики («зависание» потоков))

Генерация простых чисел

```
bool TestForPrime(int val)

{    // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) && (val % factor) )
        factor ++;
    return (factor > limit);
}
```



```
C:\WINDOWS\system32\cmd.exe
C:\classfiles\PrimeSingle\Release>PrimeSingle.exe 1 20
100%
      8 primes found between 1 and 20 in 0.00 secs
C:\classfiles\PrimeSingle\Release>_
```

```
void FindPrimes(int start, int end)
{
    int range = end - start + 1;
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
            globalPrimes[gPrimesFound++] = i;
        ShowProgress(i, range);
    }
}
```

Задание 1.

Выполнить запуски последовательной версии первоначального кода (проект Simple_number)

- Установить однопоточный режим работы (Visual Studio, Project properties -> C++ -> code generation -> Single Threaded Debug DLL)
- Выполнить компиляцию с помощью Intel C++
- Выполнить несколько запусков с различными диапазонами поиска простых чисел (start, end)

2

9

Методика разработки

Анализ

- Определить участок кода с максимальной долей вычислений

Проектирование (включить многопоточность)

- Определить, каким образом может быть использована многопоточность

Тестирование правильности работы

- Выявить источники ошибок, связанных с потоками

Измерение производительности

- Достигнуть максимальной производительности работы многопоточного приложения

З

0

Рабочий цикл

3
1

Анализ – «Sampling» («сэмплирование»)

С помощью VTune Sampling необходимо определить «узкие места» приложения

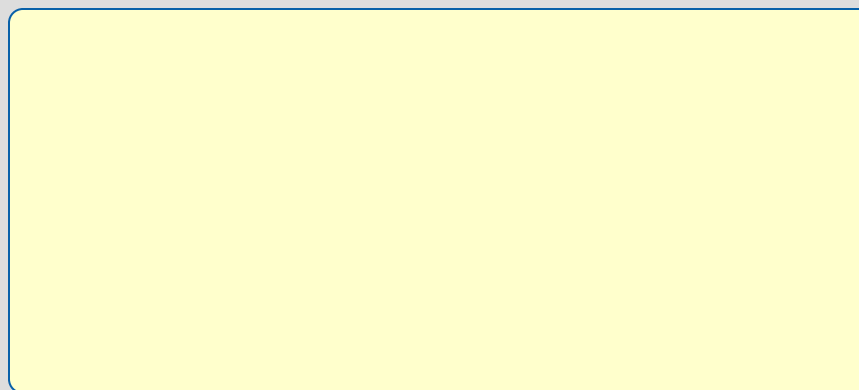
(где сосредоточена максимальная «тяжесть» вычислений)

Провести анализ работы проекта Simple_number

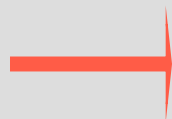
Входные данные: `start = 3` `end = 1000000`

Цель: выделить области кода, выполнение которых занимает максимальное время

Анализ – «Sampling» («сэмплирование»)



Анализ – «Sampling» («сэмплирование»)



3
4

Анализ - Call Graph

3
5

Анализ

Параллельная работа потоков будет эффективна в

- FindPrimes()

Аргументы в пользу распараллеливания

- Мало внутренних взаимозависимостей
- Возможен параллелизм по данным
- Занимает более 95% всего времени работы приложения

3

6

Задание 2

- Выполните запуск с параметрами `1 5000000` (границы диапазона поиска простых чисел)

Цель запуска - получить значение времени, с которым будет сравниваться время работы многопоточного приложения

Мы прошли первый этап цикла разработки:

- Анализ последовательного кода с помощью VTune
- Выявление функций с максимальным временем работы – «узких мест»

3

7

Метод проектирования Фостера

Необходимо выполнить 4 шага:

- **Разбить задачу на максимальное число подзадач**
 - **Установить связи «данные - вычисления»**
 - **«Агломерация»:**
составить задания, которые можно выполнять параллельно
 - **«Распределение» - распределить задания между процессорами/потоками**
- 8

Проектирование многопоточной программы

«Дробление»

- Разбить исходную задачу на подзадачи

«Связи»

- Определить типы и количество связей между подзадачами

«Агломерация»

- Составить задания – сгруппировать «мелкие» подзадачи в «большие» группы – по принципу минимума связей между группами

«Распределение»

- Распределить задания между процессорами/потоками



**Многопоточная
программа**

Модели параллельного программирования

Функциональная декомпозиция

- Параллельное выполнение разных подзадач
- Разделение на различные подзадачи, но обработка общих данных каждой подзадачей
- Выделение независимых подзадач для распределения между процессорами/потоками

Декомпозиция по данным

- Выделение операций, общих для различных данных
- Разбиение данных на блоки, которые можно обрабатывать независимо

Способы декомпозиции

Функциональная декомпозиция

- Сфокусирована на методах обработки данных, выявляя структуру задачи

Аналогии для функциональной декомпозиции и декомпозиции по данным

Независимые этапы вычислений

Функциональная декомпозиция

Задача потока связана со «стадией вычислений»

- Аналогия с конвейером сборки автомобиля – каждый рабочий(поток) параллельно с другими собирает все детали одного (своего) типа – затем общая сборка автомобиля

Декомпозиция по данным

- **4** Поточковый процесс выполняет все стадии для своего блока данных
- **2** Каждый рабочий собирает свой автомобиль

Проектирование

Ожидаемый выигрыш

$$\text{Ускорение}(2P) = 100 / (96/2 + 4) = \sim 1.92X$$

Как бы его достичь минимальными усилиями?

Параллелизм – с помощью OpenMP !

Долго ли это - распараллелить?

Сразу получится – или «путем итераций»?

4

3

OpenMP

«Вилочный» параллелизм:

- «Мастер» - поток создает команду потоков
- Последовательная программа преобразуется в параллельную

4

4

Проектирование

```
#pragma omp parallel for
    for( int i = start; i <= end; i+= 2 ){
        if( TestForPrime(i) )
            globalPrimes[gPrimesFound++] = i;
        ShowProgress(i, range);
    }
```

4

5

Задание 3

Выполнить запуск версии кода с OpenMP

- Включите библиотеки OpenMP и установите многопоточный режим MultyThreaded Debug DLL
- Выполните компиляцию
- Запуск с параметрами `1 5000000` для сравнения
- Определите ускорение

4

6

Проектирование

А каков был ожидаемый выигрыш?

А как его достичь ?

Ускорение 1.40X (меньше 1.92X)

А как долго ?

А сколько попыток ?

А возможно ли ?

**4
7**

Тестирование правильности работы программы по ее результатам

4

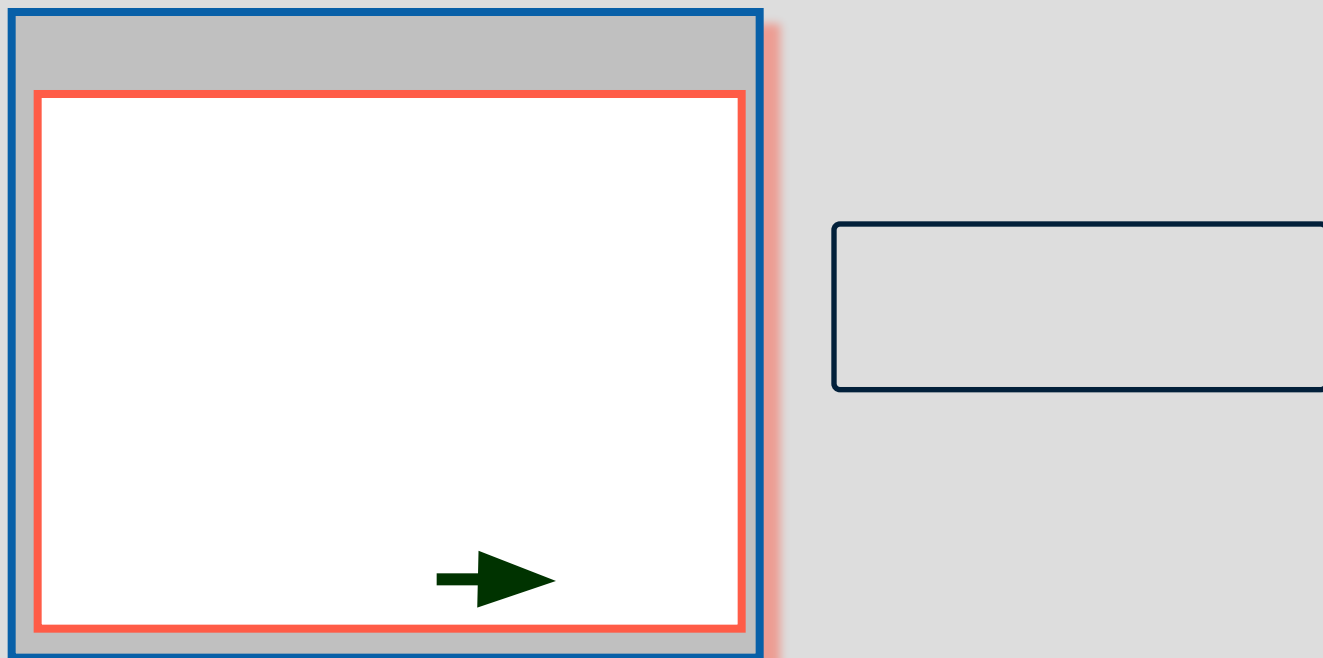
Результаты неправильные

8

Каждый запуск – свой результат...

Тестирование правильности работы

Intel® Thread Checker может определить ошибки типа «гонки данных» или «конфликты запись-запись, чтение - запись»



Thread Checker – окно результатов после выполнения анализа

1st Access[Line]

5
0

Thread Checker



Двойной щелчок
«мыши» - находим
локализацию ошибки
в коде



5
1

Thread Checker – локализация ошибки в коде

PrimeOpenMP.cpp Thread Checker - Activity

Memory write at "PrimeOpenMP.cpp":110 conflicts with a prior memory write at "PrimeOpenMP.cpp":110 (output dependence)

1st Access

Location of the first thread that was executing at the time the conflict occurred

Stack:

- ?FindPrimes@@YAXHH@Z
- "PrimeOpenMP.cpp":110
- [PrimeOpenMP.exe, 0x1293]
- ?FindPrimes@@YAXHH@Z
- "PrimeOpenMP.cpp":106
- [PrimeOpenMP.exe, 0x119c]
- main
- "PrimeOpenMP.cpp":130
- [PrimeOpenMP.exe, 0x1092]

Address	Line		Source
0x1161	104		int range = end - start + 1;
	105		
0x1170	106	?	#pragma omp parallel for
0x120E	107		for(int i = start; i <= end; i += 2)
	108		
0x1276	109		if(TestForPrime(i))
0x1284	110	x	globalPrimes[gPrimesFound++] = i;
	111		
0x1299	112		ShowProgress(i, range);
	113		}
0x11E3	114		}
	115		

2nd Access

Location of the second thread that was executing at the time the conflict occurred

Stack:

- ?FindPrimes@@YAXHH@Z
- "PrimeOpenMP.cpp":110
- [PrimeOpenMP.exe, 0x1293]
- ?FindPrimes@@YAXHH@Z
- "PrimeOpenMP.cpp":106
- [PrimeOpenMP.exe, 0x119c]
- main
- "PrimeOpenMP.cpp":130
- [PrimeOpenMP.exe, 0x1092]

Address	Line		Source
0x1161	104		int range = end - start + 1;
	105		
0x1170	106	?	#pragma omp parallel for
0x120E	107		for(int i = start; i <= end; i += 2)
	108		{
0x1276	109		if(TestForPrime(i))
0x1284	110	x	globalPrimes[gPrimesFound++] = i;
	111		
0x1299	112		ShowProgress(i, range);
	113		}
0x11E3	114		}
	115		

Diagnostics Stack Traces Source View

Задание 4

Примените Thread Checker для анализа правильности выполнения

- Создать Thread Checker activity
- Запуск приложения с параметрами 3 20
- Есть ошибки ?

5
3

Тестирование правильности работы

Сколько попыток еще предпринять?

**Thread Checker обнаружил
ошибку, значит, еще трудиться
и трудиться...**

Как долго трудиться над этим распараллеливанием?

5

4

Тестирование правильности работы

```
#pragma omp parallel for
    for( int i = start; i <= end; i+= 2 ){
        if( TestForPrime(i) )
#pragma omp critical
            globalPrimes[gPrimesFound++] = i;
        //ShowProgress(i, range);
    }
```

5

5

Задание 5

Модифицируйте версию кода с OpenMP

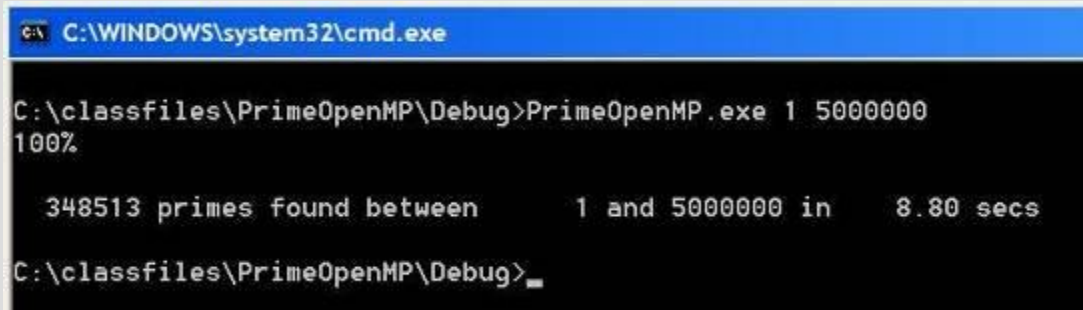
- Добавьте прагму критической секции в код
- Откомпилируйте код
- Проверьте Thread Checker
 - Если будут ошибки, исправьте их, и снова выполните проверку Thread Checker
- Запуск `1 5000000` для сравнения
 - Проверьте Thread Checker
 - Ускорение ?

5

6

Correctness

Работает-то правильно, да ускорение низкое...~**1.33X**



```
C:\WINDOWS\system32\cmd.exe
C:\classfiles\PrimeOpenMP\Debug>PrimeOpenMP.exe 1 5000000
100%
348513 primes found between 1 and 5000000 in 8.80 secs
C:\classfiles\PrimeOpenMP\Debug>
```

Разве это предел, к которому мы стремились?

Нет! По закону Амдала мы можем достичь ускорения 1.9X

Задачи повышения производительности

Параллельный «оверхед» (overhead)

- «*Накладные расходы*» на создание потоков, организацию «расписания» их работы ...

Синхронизация

- Применение без особой необходимости *глобальных переменных*, которые автоматически являются объектами синхронизации для всех потоков –

если один поток изменил значение глобальной переменной, значит, работа остальных будет приостановлена до тех пор, пока каждый поток не «установит у себя» новое значение глобальной переменной

Дисбаланс загрузки

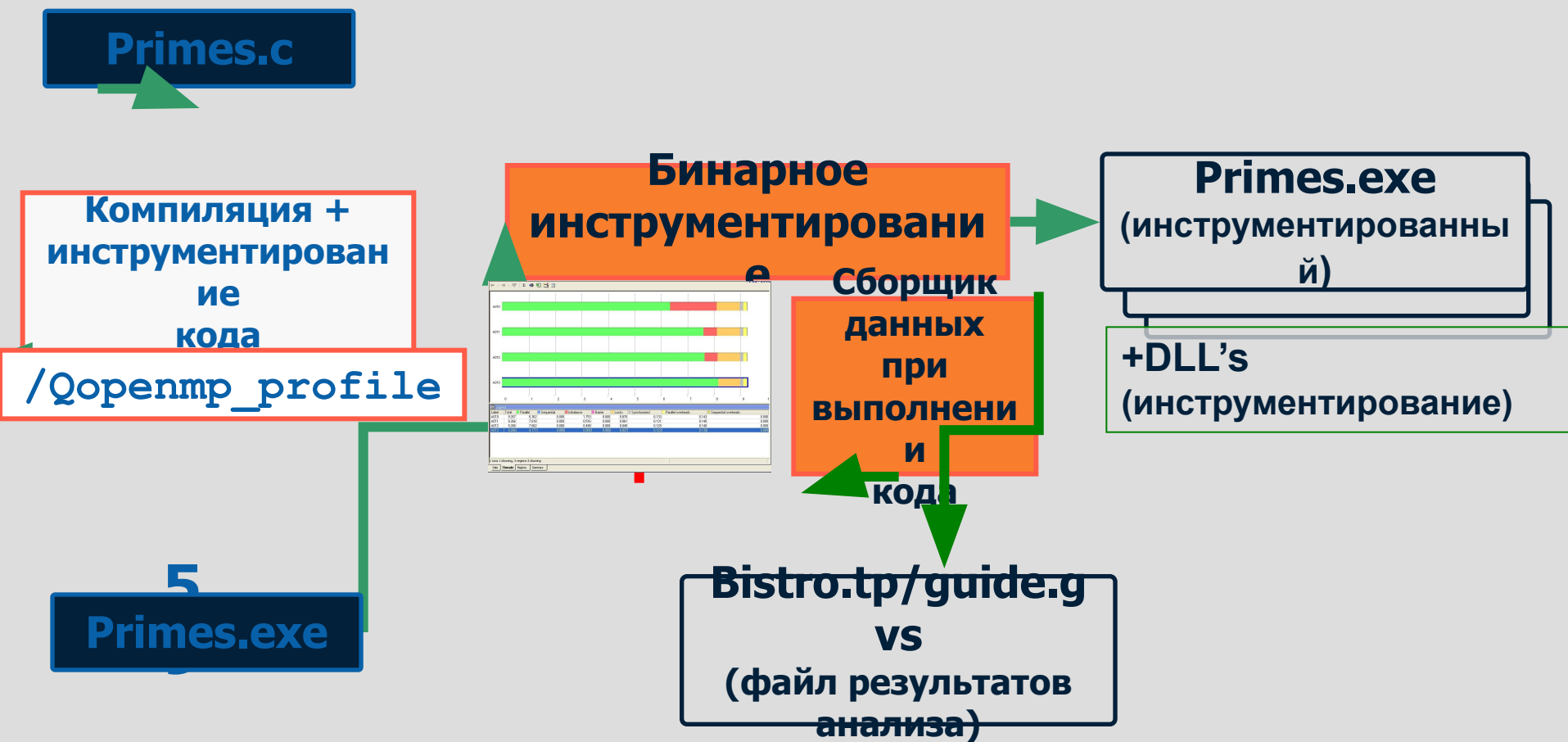
- Недостаточно эффективное распределение работы между потоками «*кому сколько работать*» - один свою работу сделал и ждет, а другие работают....

Гранулярность

- Б распределение «квантов» работы для потоков в пределах одного параллельного региона (все потоки выполняют свой «квант» - затем «хватывают» следующий) – должно решать проблему дисбаланса загрузки

Измерение производительности

Thread Profiler определяет «узкие места» - участки кода многопоточной области, где работа потоков происходит «с низким КПД»



Thread Profiler for OpenMP

- Только для OpenMP приложений
- Окно результатов «Summary» - появляется сразу после завершения анализа Thread Profiler
- Стрелками показана расшифровка цветовой диагностики

Thread Profiler for OpenMP

6
1

Thread Profiler for OpenMP

6
2



Thread Profiler for OpenMP

6
3

Thread Profiler for OpenMP

- Окно «Regions»: регионы – область кода программы, либо полностью последовательного, либо полностью параллельного (параллельный регион)
- показывает время работы каждого участка (например, дисбаланс, барьер – «цветовые участки») для каждой области (региона) кода

Thread Profiler for OpenMP

6
5



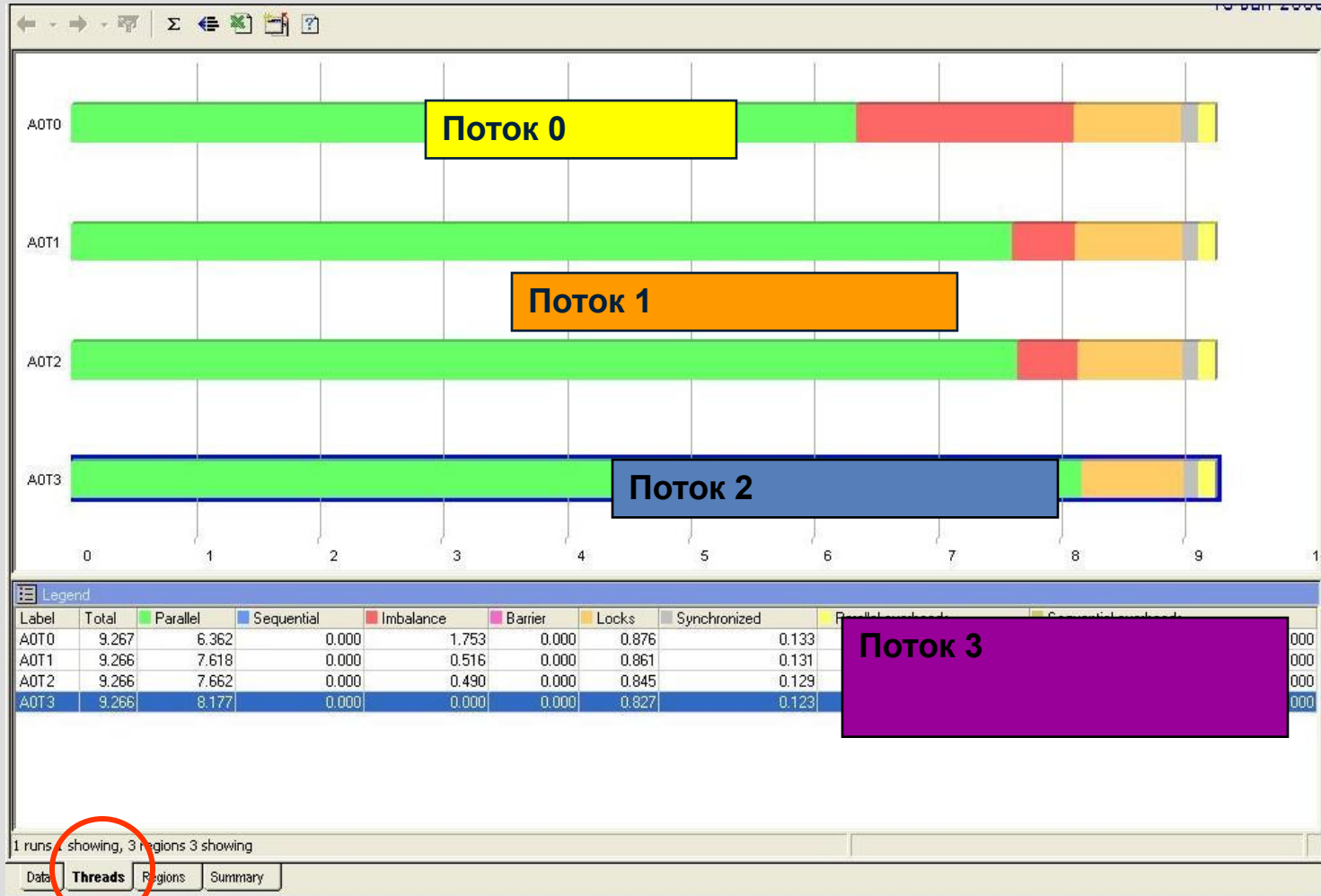
Задание 6.

- Исследуйте параллельную работу программы Thread Profiler с теми же параметрами, что и базовое измерение
- Число потоков установите, равное 4

6

6

Диагностика Thread Profiler – большой дисбаланс – потоки «ждали друг друга»



Определили дисбаланс загрузки

- Распределим работу более эффективно: не по $\frac{1}{4}$ от всего цикла сразу каждому потоку, «пока не встретимся», а каждому по несколько итераций цикла, «пока не встретимся», затем – новые несколько итераций

```
void FindPrimes(int start, int end)
{
    // start is always odd
    int range = end - start + 1;

#pragma omp parallel for schedule(static, 8)
    for( int i = start; i <= end; i += 2 )
    {
        if( TestForPrime(i) )
#pragma omp critical
            globalPrimes[gPrimesFound++] = i;
        //ShowProgress(i, range); }
    }
```

6

8

Борьба с дисбалансом – перераспределение заданий потокам

- Новое «распределение работы» по сравнению со старым будет следующим образом выглядеть в графическом представлении:



Задание 7

Для уменьшения дисбаланса

- установить `schedule (static, 8)` «клаузу» OpenMP для `parallel for pragma`
- Запуск
- Ускорение?

7
0